View-Oriented Transactional Memory

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

***Abstract*—This paper proposes the View-Oriented Transactional Memory (VOTM) model to seamlessly integrate locking mechanism and transactional memory. The VOTM model allows programmers to partition the shared memory into "views", which are non-overlapping sets of shared data objects. The Restricted Admission Control (RAC) scheme can then control the number of processes accessing each view individually 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. Experimental results demonstrate that VOTM outperforms traditional transactional memory models such as TinySTM by up to three times.**

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

I. 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 [1, 2].

Traditionally locking [3, 4] is used for concurrency control, where multiple processes/threads[1] have to access a shared data object in an exclusive way. Atomic access to a shared object is achieved through a locking mechanism. This lock-based concurrency control is generally regarded as pessimistic approach [5] 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 processes. 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) [6, 7] was proposed for shared-memory programming models. In TM, atomic access to shared objects is achieved through transactions. All processes 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 processes, 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 [8, 9] 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, the TM-based approach has little roll-back overhead and encourages high concurrency since multiple processes 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, 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 this paper, we propose the novel View-Oriented Transactional Memory (VOTM) paradigm that seamlessly integrates the locking mechanism and transactional memory into the same programming model. VOTM is designed based on the generic principle of our previous View-Oriented Parallel Programming (VOPP) model [10–12]. 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

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

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 [13] 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 the novel Restricted Admission Control (RAC) scheme that can dynamically decide if the locking mechanism or a transaction should be used for the access of a view and thus seamlessly integrates the merits of both the lock-based and the TM-based approaches.

In the RAC scheme, a set of shared objects (grouped as a view using either static declaration or dynamic memory allocation in VOTM) is restricted to be accessed by a limited number of processes Q (called admission quota) whose value ranges from 1 to the maximum number of processes (*NPROCS*), depending on the contention between the processes. The limited number of processes can be statically specified in the program or dynamically adjusted at runtime according to the contention situation, e.g., the number of transactional aborts. When Q is 1, the processes access the set of data objects sequentially as in the lock-based approach. When Q equals *NPROCS*, the RAC scheme behaves like the TM-based approach where any process is allowed to start a transaction to access the data objects. However, when Q is greater than 1 but smaller than *NPROCS*, only Q processes are allowed to access the data objects concurrently through transactions. If there are already Q processes accessing the data objects inside uncommitted transactions, other processes are excluded from accessing the set of data objects and have to wait until some existing transactions commit. Furthermore, RAC can flexibly adjust Q at runtime in order to achieve optimal performance, which will be described in details in Section II-B.

A. Contributions of this paper

First, we usher in a new programming paradigm VOTM, which enables programmers to achieve optimal performance based on view partitioning while avoiding problems from lock-based programming such as fine-grained locking and deadlock. Complex data structures such as linked lists, trees, and graphs can be simply placed into different views, but efficient access to them is achieved through RAC.

Second, 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 each view.

Third, we demonstrate the performance of VOTM is much better than traditional TM and lock-based approaches through experimental results.

The rest of the paper is organized as follows: Section II will present the details of the VOTM model, the RAC scheme, and the implementation; Section III will show experimental results and performance evaluation; Section IV will discuss related work and Section V concludes the paper.

II. The VOTM programming model and implementation

VOTM is based on the philosophy of shared memory partitioning. Since different shared data can have different access patterns and contention levels, VOTM allows groups of shared objects that are not required to be accessed atomically to be put into different views, so that concurrency control on each view can be separately optimized using the RAC scheme (refer to Section II-B for more details).

This optimization cannot be achieved by traditional transactional memory without grouping data objects into views. For example, in VOTM a tree structure with thousands of nodes can be put into one view, and a hash table can be put into another view if they are not required to be accessed atomically at the same time in an application. Suppose the tree in the application has high contention, but the hash table has low contention. The RAC scheme in VOTM would quickly restrict the access to the tree to relieve its contention, without restricting the number of processes accessing the hash table. In this way, the system would continue to allow maximal concurrent access to the hash table, though the access to the highly-contentious tree is restricted. Therefore, by putting the tree and the hashtable in different views, their accesses are separately optimized, which cannot be achieved by traditional transactional memory.

In addition, the RAC scheme also allows seamless integration of the locking mechanism and transactional memory into the VOTM model. RAC can dynamically control the admission quota Q of a view, or alternatively, Q can be manually specified during view allocation. When Q is greater than 1, a transaction starts when the process is admitted to the view. However, when Q becomes 1, it is equivalent to the lock mechanism, thus eliminating all transactional overheads. In this way, programmers only need to partition shared data into views according to the access patterns, but leave concurrency control to RAC.

A. 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 for the linked list, and *malloc_block()* allocates a memory block from the view.

A VOTM code snippet for list insertion is shown in Figure 2. Here the parameter *node* in the function points to a node that is a memory block belonging to the view

```
typedef struct Node_rec Node;

struct Node_rec {
  Node *next;
  Elem val;
};

typedef struct List_rec {
  Node *head;
} List;

List *ll_alloc(vid_type vid) {
  List *result;
  create_view(vid, size, 0);
  result = malloc_block(vid, sizeof(result[0]));
  acquire_view(vid);
  result->head = NULL;
  release_view(vid);
  return result;
}
```

Figure 1. Code snippet of list allocation in VOTM

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.

```
void ll_insert(List *list, Node *node, vid_type vid) {
  Node *curr;
  Node *next;

  acquire_view(vid);

  if (list->head->val >= node->val) {
    /* insert node at head */
    node->next = list->head;
    list->head = node;
  } else {
    /* find the right place */
    curr=list->head;
    while (NULL != (next = curr->next) &&
           next->val < node->val) {
      curr = curr->next;
    }
    /* now insert */
    node->next = next;
    curr->next = node;
  }
  release_view(vid);
}
```

Figure 2. Code snippet of list insertion in VOTM

Deadlock can be avoided in VOTM if view acquisitions are not nested. If two views need to be acquired in a nested way, they can often be either put into the same view initially or merged together dynamically. If views are carefully partitioned, nested view acquisitions are rarely needed in real applications. When nested view acquisitions are needed, they can often be resolved in VOTM by merging the involved views into one view.

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

B. Restricted Admission Control (RAC) scheme

We implement the RAC scheme for every view. 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 processes 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 processes 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 process 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 processes (*NPROCS*) if it is not set statically at view creation time. RAC regularly checks the contention situation of the view. The contention situation is indicated by the number of aborts as well as the number of successfully committed transactions that are related to the view. If the number of aborts is high, the contention is usually high. However, high number of successful transactions often indicates that the contention is not high enough to affect the overall progress of the computation, even though the number of aborts may be high in such a situation. Therefore, we use the ratio between the number of aborts and the number of successful transactions ($aborts/successful_tx$) to reflect the severity of the contention situation.

If this abort/success ratio is larger than MAX (currently set to 8.0), the view is considered as highly contentious. When this happens, RAC will relieve the contention of the view by halving the admission quota Q of the view. Then, the number of aborts and the number of successful transactions will be reset in the view. This process can be repeated periodically until Q reaches 1, in which case the concurrency control is switched to the lock-based approach. Then, the transaction mechanism is no longer used to access the view and the abort/success ratio for the view concerned is no longer checked.

Conversely, when Q is greater than 1 and the abort/success

Table I
VOTM API

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*.

ratio is smaller than MIN (currently set to $1/8$), the view is considered as having low contention. Then, RAC will increase concurrency by doubling Q. When Q is changed, the numbers of aborts and successful transactions of the view will be reset. This process will repeat periodically until Q reaches *NPROCS*.

To eliminate cache flushing overheads on incrementing the shared counters in the RAC metadata of the view, when it is clear that access restriction to the view is unnecessary because of following low contention condition:

- 20000 transactions are executed since Q is set to $NPROCS$, and
- the abort/success ratio $< MIN$

The RAC mechanism will be disabled.

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.

The choices of MAX and MIN are currently empirical. Different TM algorithms may favor different values. For example, the encounter-time locking TM algorithm used in TinySTM aborts potentially-conflicting transactions early to reduce wasted computation. Under the same contention situation, this would result in higher abort/success ratio than other TM algorithms such as commit-time locking used in TL-2. Therefore, the same genuine high contention case will have higher abort/success ratio for TinySTM than for TL-2. Optimal MAX and MIN settings are dependent on the underlying transactional memory system. Automatic adjustment of these values is an interesting issue for further research.

Frequent check of the abort/success ratio is costly since a spinlock is used for multiple processes to access the numbers, which would significantly increase the overhead of RAC. Therefore, the periodic check is only triggered under the condition when the sum of aborts and successful transactions is a multiple of 5000. Our observations show that, checking under this condition is frequent enough in most cases, because if the contention is high, the number of aborts will rise quickly to trigger the check.

C. Implementation details

We implement the VOTM model based on the software transactional memory system TinySTM [14], a word-granularity timestamp-based TM system based on the C language. The algorithm of TinySTM is based on the lazy snapshot algorithm (LSA) [15]. In our implementation, TinySTM is configured as a redo-log-based TM system with encounter-time locking.

In VOTM, access to each view can be controlled independently so that a view with high contention will not affect concurrency of other views that may have low contention. Experimental results in the next section demonstrate that using multiple views in this way improves performance.

Similar to TinySTM and many other software TM systems, in our current implementation, the memory accesses in VOTM have to be explicitly labelled with primitives such as *Tx_read* and *Tx_write*. However, these primitives can be removed with compiler support or hardware TM systems [16].

Since we use encounter-time locking, the transaction first writing to a location commonly accessed by other transactions wins (as opposed to TL-2, which uses commit-time locking instead). However, no matter what conflict detection policy is used, short transactions can easily abort a long transaction and computation done by the long transaction will be wasted. This situation will be further explained in Section III-A.

D. Origin of performance gain in VOTM

The origin of performance gain in VOTM is very different from TM systems that use either in-transaction conflict resolution algorithms and/or transaction scheduling algorithms. In-transaction conflict resolution algorithms [17–19] only detect conflicts and control contentions during the execution of transactions and on their own still allow any processes to freely enter transactions. Transaction scheduling algorithms [20–22] prevent conflicts by serializing transactions or limiting the number of concurrent transactions. These algorithms treat the entire TM with the same scheduling decision. However, it is not reasonable to restrict access

to a low-contention shared object due to another shared object that has high contention, a situation that could happen on these algorithms. In VOTM, transactional memory is divided into views where shared objects that will be accessed together in a transaction are grouped into the same view. In this way, restricting access to a view with high contention does not affect access to a view with low contention, which enables more concurrency. In VOTM, RAC is used as the transactional scheduling algorithm for each view, but any in-transaction conflict resolution algorithms can be applied in each view. Regardless of the choice of the underlying in-transaction conflict resolution algorithm, there are always cases where the number of aborts becomes very high. Here RAC can reduce contention by limiting the admission of processes to the view, and improve progress. We will further discuss transactional scheduling technologies in Section IV.

In the next section, we will show that VOTM with RAC can reduce the number of aborts, and therefore reduce contention and increase throughput, by controlling the admission to each view.

III. Performance evaluation

In this section, we compare the performance of VOTM with the software transactional memory system TinySTM version 1.0.0 [14] and the lock-based approach which uses Pthreads mutexes. Our benchmark applications include Bayes, Intruder, Genome, Labyrinth, Vacation and SSCA2 from the STAMP transactional memory benchmark suite version 0.9.10 [23], and Travelling Salesman Problem (TSP) from the SPLASH-2 benchmark suite [24]. They represent different classes of applications. The experiments 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. The runtime calculated in each case includes initialization and finalization costs. However, the runtime of functions that are irrelevant to the original application, such as generation of random input sequences and result-verification, is excluded.

Intruder has short transactions with high contention. In this application, a dictionary is used to store partial results, and jobs are handled by a centralized task queue. In the VOTM version, the task queue and the dictionary are allocated in separate views.

In Bayes, the shared net is accessed by long transactions with high contention, whereas access to the task queue is short and does not take computation time. Since the net is never accessed together atomically with the task queue, they are allocated in separate views. Default parameters "-v32 -r4096 -n10 -p40 -i2 -e8 -s1" are used.

Genome is a gene-sequence alignment algorithm which has multiple shared hash tables with low contention and two shared arrays with higher contention. Shared data structures include an input hash table as well as an array of hash tables containing intermediate fragments plus two arrays tracking prefixes and suffixes. In the VOTM version, a view is used to host all shared data structures. In the pure lock-based version, the hashtable, prefix array, and suffix array are each protected by a lock. We could protect each bucket in each hash table with a lock, but this would be too tedious and change the original algorithm drastically. Default parameters "-g16384 -s64 -n16777216" are used.

Both Labyrinth and Vacation have long transactions with little contention. Labyrinth finds the shortest path between pairs of starting and ending points in a maze, which is implemented as a shared grid. The shared grid is accessed with long transactions with low contention. The input file "random-x512-y512-z7-n512.txt" is used. The shared grid is allocated as a view in the VOTM version. Since access to the grid cannot be divided without a complete rewrite of the algorithm, the pure lock-based version simply uses lock to protect access to the grid.

Vacation simulates the operation of a travel agency manager. Each transaction consists a set of operations including adding/removing reservations. The transaction succeeds only if all operations succeed; otherwise, it will abort and restart. Transactions are long and with a moderately high memory accesses, but with low contention. Since all shared data can be accessed together atomically, they must be put into a single view in the VOTM version. Also for the same reason, the critical section cannot be broken down in the pure lock-based version; therefore, a single lock is used to protect the critical section. Default parameters "-n4 -q60 -u90 -r1048576 -t4194304" are used.

SSCA2 has high number of very short transactions with low contention; therefore, it serves as a test case testing overheads for starting and ending transactions. SSCA2 operates on a large, directed and weighted multigraph. Kernel 1 in this application is used in STAMP, which constructs the graph data structure in parallel using adjacency arrays and auxiliary arrays. Similar to Labyrinth, the graph in SSCA2 is put in a single view in the VOTM version. In SSCA2, operation on each graph node is done by a very short transaction that takes little computation time. Contention is very low in SSCA2 because the large number of graph node means concurrent updates on the same adjacency list is rare. However there are many transactions in this application. Default parameters "-s20 -i1.0 -u1.0 -l3 -p3" are used.

The Travelling-Salesman Problem (TSP) algorithm have short transactions with very high contention. Transactions in this algorithm are memory intensive but does not have computational work; therefore, only a small portion of execution time is spent in transactions. The algorithm uses the branch-and-bound depth-limited search approach. The 33-city case ftv33.atsp from TSPLIB95 [25] is used. In this algorithm, the priority queue (storing partially-evaluated

tours) is the shared object, and is allocated in a view. Since access to this view is short but contentious, a VOTM version with the Q manually set to 1 is also implemented to test the benefit of manual Q optimization against the VOTM version with dynamic Q adjusted by RAC.

Kmeans and Yada from the STAMP benchmark are excluded from this paper because, in Kmeans, the incrementation of each element in the shared array is atomic, so atomic operation should be used instead of TM. The Yada application crashes frequently whenever it runs with multiple processes, and when it does not crash, parallelization shows little performance gain, if any, because all computation time is spent in transactions with extremely high contention.

Table II
APPLICATION RUNTIME (s) AT $N = 16$

Application	VOTM	TinySTM	Lock-based
TSP $Q = 1$	52.23	194.73	52.23
Intruder	43.05	127.70	100.62
Bayes	11.15	19.51	30.72
Genome	4.93	5.91	37.48
Labyrinth	35.60	35.08	331.28
Vacation	14.84	14.1	61.88
SSCA2	8.80	8.77	56.28

Table III
NUMBER OF TRANSACTIONS AND ABORTS AT $N = 16$

Application	#transactions	VOTM	TinySTM
TSP $Q = 1$	3,925,092	0	4,150,852,440
Intruder	23,428,141	10,986,905	1,238,254,062
Bayes	1,751	4,591	522,972
Genome	2,472,907	83,273	64,595,381
Labyrinth	1056	196	202
Vacation	4,194,304	1,443	1,059
SSCA2	22,362,292	62	64

From Table II, it can be seen that VOTM has superior performance over TinySTM in high contention applications. In TSP and Intruder, VOTM is 270% and 200% faster than TinySTM respectively. In Bayes and Genome, VOTM is also 75% and 20% faster than TinySTM, respectively.

In the above applications, RAC successfully prevents speedup degradation by restricting the number of processes admitted to a view. In TSP, RAC eliminates aborts in VOTM altogether, and for the rest of the applications shown in Table III, RAC cuts the number of aborts in VOTM by up to 100 times. The reasons RAC improves performance of VOTM will be discussed in detail in Section III-A.

In low contention applications, such as Labyrinth and Vacation with long transactions and SSCA2 with a high number of very short transactions, the runtime of VOTM and TinySTM are similar.

At low contention, RAC will allow admission of all processes in order to maximize concurrency, and will thus behave like traditional TM. The runtimes of VOTM and TinySTM for these applications shown in Table II are similar, suggesting that VOTM has little extra overhead.

However, the pure lock-based version in general has poor performance because the applications Intruder, Bayes, Genome, Labyrinth and Vacation have coarse-grained critical sections occupying the majority of the execution time, thus eliminating concurrency. To make them work with fine-grained locking requires tedious algorithms and programming if not impossible.

Although in SSCA2, the time spent in critical sections is very short, the sheer number of acquires of the same lock (22 million acquires) in the pure lock-based version makes the lock a hot-spot. The resultant CPU cache coherence overhead makes the pure lock-based version unscalable.

Table IV
PERFORMANCE OF TSP AT $N = 16$

	VOTM (dynamic Q)	VOTM ($Q = 1$)	TinySTM	Lock-based
time(s)	71.54	52.23	194.73	52.23
#aborts	15,658,595	0	4,150,852,440	0

The application TSP has a shared priority queue with high contention. Therefore, TinySTM is not scalable. Since access to the priority queue is known to be very short but memory-intensive, VOTM benefits from manually setting Q to 1 to avoid transactional overheads. As shown in Table IV, VOTM with $Q = 1$ has a 27% performance gain over VOTM with dynamic Q. By manually setting $Q = 1$, the performance of VOTM now matches the lock-based version, because locking is more effective to protect highly contentious shared data such as this priority queue.

The above results show our VOTM model has the performance advantage over TM in high contention situations and allows the performance benefit of fine-grained locking through the optimization of admission quota.

A. How RAC improves performance

RAC improves performance in two ways. The first way is through removing the transactional overhead by switching to lock-based mechanism when the admission quota Q equals 1.

To investigate this transactional overhead, microbenchmarks of transactions with 0, 1, 10, 100, 1000, 10000 and 100000 read and write operations are performed. Each read/write operation is performed in a *separate* location to examine the real cost of read- and write-set maintenance. To amortize measuring noise, we have collected the results by first measuring the execution time of 100,000 sequentially-executed identical transactions and then calculating the average execution time of one transaction. The results are presented in Table V.

Table V
OVERHEAD OF TRANSACTIONS WITH DIFFERENT SIZE

no. of r/w	0	1	10	100	1000	10000	100000
time(μs)	0.21	0.35	1.30	10.65	109.47	1216.22	14425.03

From Table V, it can be seen that the cost of starting and ending a transaction itself is not trivial ($0.21\mu s$ per empty transaction), and for a long transaction with 100,000 reads and 100,000 writes, the overhead can be up to $14ms$ per transaction. Therefore, transactions are expensive.

To avoid the expenses in transactional memory, RAC drops the transactional memory mechanism when the admission quota of a view becomes 1.

The second way that RAC improves performance is through reducing the number of aborts by decreasing Q. As the application is run, the RAC algorithm adjusts Q according to the abort/success ratio. Q will eventually settle at the value where access speedup saturates (i.e. the number of processes where maximum concurrency is reached).

After the speedup of accessing the view is saturated, RAC prevents speedup degradation by restricting admission to the view to Q processes to prevent extra processes from increasing contention and conflicts. This is very important in real-life situations, as it can be difficult to determine in advance the number of processes needed to saturate access speedup if the access patterns are dynamic and bursty.

In order to demonstrate the effect of RAC in terms of restricted admission, we use Bayes in this part of the experiment. Here the number of running processes (N) is fixed to 16 and the admission quota (Q) is fixed to 1, 2, 4, 8 and 16 respectively. The $Q = 16$ case is equivalent to no restriction of admissions, but the $Q = 1$ case still uses transactions (tx) in order to show only the effect of admission control. However, result of a $Q = 1$ case run without transactions (no tx) is also shown to demonstrate transactional overheads.

Table VI
RUNTIME AND NUMBER OF ABORTS OF BAYES AT DIFFERENT Q

	1(no tx)	1(tx)	2	4	8	16
time(s)	27.51	28.34	23.53	12.42	9.4	12.54
#aborts	0	0	337	1143	3422	536384

From Table VI, it can be seen that Bayes performs the best at $Q = 8$. When Q is smaller, the performance is not good due to lack of concurrency, though the number of aborts is small. However, when Q is larger, the performance gets worse due to high contention. Therefore, RAC is very useful for adjusting Q to the optimal value. Differences between $Q = 1$ cases with and without using transactional mechanisms reflect transactional overheads.

Figure 3 shows a scenario explaining theoretically why RAC can improve performance with restricted admission. As mentioned earlier, in TinySTM, a late-coming short transaction can easily abort a long transaction that has been running for a long time if the short transaction locks an object first. The time between the entry of the long transaction and the short transaction will be wasted. RAC can reduce the likelihood of this situation by restricting the number of processes acquiring the view.

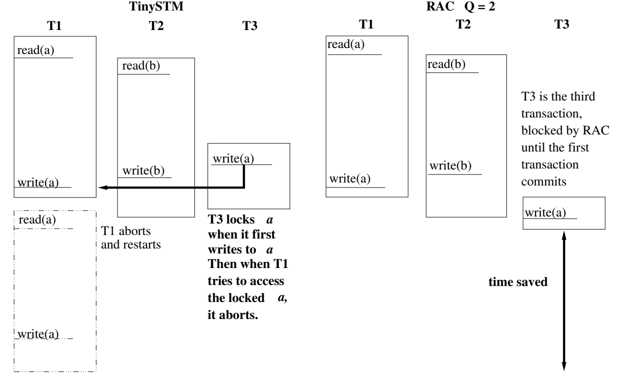

Figure 3. RAC implementation over TinySTM - RAC blocks T3 and prevents it from aborting T1 in high contention

In Figure 3, the long transaction *T1* conflicts with the short transaction *T3*, although *T3* starts much later than *T1*, *T3* locks the variable *a* first. T1 finds out the conflict when it tries to write to the variable *a*, then it aborts and restarts. However, if Q is set to 2 by RAC, *T3* is the third transaction to begin, so it is blocked until the first transaction (*T1*) commits, which prevents it from conflicting and aborting *T1*.

The above results and analysis have demonstrated the advantage of RAC that can dynamically adjust the admission quota Q to the optimal, keeping the best balance between concurrency and contention.

B. View partitioning improves performance

To investigate benefits of view partitioning over traditional TM with transaction scheduling, performance of VOTM and "TinySTM + RAC" is compared using the applications Intruder and Bayes. "TinySTM + RAC" is a system that implements the transaction scheduling algorithms like RAC for the entire TM.

Table VII
PERFORMANCE OF VOTM AND TINYSTM + RAC AT $N = 16$

	Application	VOTM	TinySTM + RAC
time(s)	Bayes	11.15	11.97
	Intruder	43.05	59.50
#aborts	Bayes	4591	4587
	Intruder	10986905	10337777

Table VII shows that for Intruder and Bayes where their VOTM versions have multiple views, VOTM outperforms

"TinySTM + RAC" by 38% and 7% respectively. In these applications, both VOTM and "TinySTM + RAC" experience similar contention.

In both applications, a view with high contention is often accessed at the same time as another view with low contention. For example, in Intruder, a process can dequeue a task from the task list (with low contention) while another process can access the dictionary, which has high contention, and therefore access is restricted by RAC in VOTM. Similarly in Bayes, the task list and the highly-contended net are allocated in separate views. By placing the task list and the high contention data, such as the dictionary and the net in separate views in VOTM, the restriction placed on access to the dictionary and the net will *not* affect access to the task list and reduce concurrency. However, in "TinySTM + RAC", the entire shared memory is restricted to access under the same admission quota. Therefore, access of all data structures in the shared memory, including the task list with little contention, will be restricted as a result of contention in the dictionary and the net. That is, the concurrency of processes accessing the task list will be unnecessarily affected in "TinySTM + RAC". As shown in the above results, the memory partitioning philosophy of VOTM resolves this problem and therefore has superior performance over transactional memory with transaction scheduling algorithms like "TinySTM + RAC".

IV. Related work

A. Transactional scheduling

All in-transaction conflict resolution algorithms, including both early-locking (such as DSTM [17, 26], SXM [18] and McRT-STM [27]) and late-locking (such as TL-2 [19] and NOrec [28]) algorithms, resolve conflicts *within* a transaction only *after* these conflicts have been detected, but processes 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.

Recently, some transaction scheduling algorithms have evolved to control *admission* of processes into transactions when contention is high, aiming at preventing conflicts before they occur and therefore reducing wasted work on aborted transactions. This family of transactional scheduling algorithms works orthogonally with the in-transaction conflict resolution algorithms mentioned above.

Transaction scheduling algorithms such as [21] use a process-local contention score. When a process experiences high contention, it queues the starting transaction to a central scheduler, which will execute queued transactions serially. [22] adopts a similar approach, except when a process 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 processes. 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 [20] also adopts a similar approach.

However, as discussed in Section II-D and Section III-B, 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 another high-contention shared object. Also the statistics collected for the entire TM are not as accurate as those collected per view basis and are thus less applicable.

B. Adaptive locks

The speculative lock elision (SLE)-based model [29] 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 processes, but only one process can hold an elidable lock non-speculatively at any time. In addition, a non-speculative process trying to acquire an elidable lock will not be blocked by other speculative processes currently holding the lock. The system keeps track of accesses to shared memory and ensures that the non-speculative atomic section always win and other conflicting speculative atomic section will be aborted. When acquiring an elidable lock or restarting an aborted atomic section, the system uses a heuristic approach to decide whether to acquire the elidable lock speculatively.

The adaptive lock model in [30] has a similar approach, except a process trying to acquire the lock in mutex mode must wait until all existing processes 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 processes to hold the elidable lock in speculative mode, or exclusive access to one thread during non-speculative (mutex) mode, yet as shown in section III-A, there are some cases where the optimal admission quota of a lock/view is actually between 1 and $NPROCS$. Therefore, RAC 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 process or admitting all processes.

V. Conclusions and future work

VOTM allows shared data with different access patterns to be allocated in different views, and then let RAC optimize

access to each view independently according to the contention level of each view. Therefore, processes accessing a view with low contention will not be hindered by restrictions placed on another view with high contention.

With RAC, VOTM seamlessly integrates locking mechanism and transactional memory into one programming paradigm. It can take advantage of the merits of both the pessimistic (locking) and the optimistic (TM) approaches to concurrency control. Programmers do not need to worry about concurrency control of the view, because concurrency control is left to the system (RAC) to decide whether a locking mechanism or a transactional mechanism should be used based on the contention situation of the view.

RAC can improve performance of VOTM regardless of which underlying TM algorithm is used. In any TM algorithms, there will be situations where the contention become very high (number of aborts becomes much larger than the number of transactions), and in these situations, RAC will step in and restrict admission to the view to control contention, thereby reduces works wasted by aborted transactions and improves progress. Experimental results show that RAC has superior performance to both TM and the lock-based approach because of the ability of RAC controlling admission and switching between TM and locking, whereas traditional 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. Therefore, through the definition/creation of different views in TM, VOTM offers better performance than traditional transactional memory and better convenience (and sometimes better performance) than lock-based programming. We believe this new programming paradigm will bridge the gap between TM and lock-based programming, and thus will bring more vitality to the research of TM.

One issue with RAC is blocking of processes by RAC when Q is smaller than *NPROCS*. This blocking seems to violate the lock-free or obstruction-free feature of TM systems [31]. Even though this feature is arguably necessary [32], RAC can quickly resolve this kind of blocking when the contention becomes low and thus Q is increased up to *NPROCS*, 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.

Another issue with the current VOTM model is the possibility of deadlock during nested view acquisition. However, in most of the cases, nested view acquisition is not necessary, as shared data that can be accessed together atomically should be allocated in the same view. For example, in VOTM, all nodes in a tree will be allocated into the same view, thus nested view acquisition for individual nodes of the tree is unnecessary.

As a future work, we will investigate potential refinements on the RAC algorithm, such as adaptive adjustment of the sampling interval, discovery of optimal parameters like *MIN* and *MAX* for the abort/success ratio, and the impact of different underlying TM algorithms (such as NOrec [28]) on these parameters. We will also benchmark VOTM against other transactional scheduling and adaptive lock systems to identify performance and overheads in different cases.

References

[1] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee, "Software transactional memory: Why is it only a research toy?" *Queue*, vol. 6, pp. 46–58, September 2008.

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

[3] L. Lamport, "A new solution of Dijkstra's concurrent programming problem," *Commun. ACM*, vol. 17, no. 8, pp. 453–455, 1974.

[4] G. Peterson, "Myths about the mutual exclusion problem," *Information Processing Letters*, vol. 12, no. 3, pp. 115–116, 1981.

[5] A. Tanenbaum and M. Steen, *Distributed Systems: Principles and Paradigms, Chapter 5*. Prentice Hall, 2002.

[6] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," *SIGARCH Computer Architecture News*, vol. 21, pp. 289–300, May 1993.

[7] D. B. Lomet, "Process structuring, synchronization, and recovery using atomic actions," in *ACM Conference on Language Design for Reliable Software*, March 1977, pp. 128–137.

[8] H. Kung and J. Robinson, "On the optimistic methods for concurrency control," *ACM Transactions on Database Systems*, vol. 6, no. 2, pp. 213–226, June 1981.

[9] P. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Computer Survey*, vol. 13, no. 2, pp. 185–221, June 1981.

[10] K.-C. Leung, Z. Huang, Q. Huang, and P. Werstein, "Data race: Tame the beast," *Journal of Supercomputing*, vol. 51, no. 3, pp. 258–278, March 2010.

[11] 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.

[12] 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.

[13] L. Ceze, P. Montesinos, C. von Praun, and J. Torrellas, “Colorama: Architectural support for data-centric synchronization,” in *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, 2007, pp. 133–134.

[14] 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.

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

[16] D. Dice, Y. Lev, M. Moir, and D. Nussbaum, “Early experience with a commercial hardware transactional memory implementation,” in *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2009, pp. 157–168.

[17] W. N. Scherer, III and M. L. Scott, “Advanced contention management for dynamic software transactional memory,” in *Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing*, M. K. Aguilera and J. Aspnes, Eds. ACM, 2005, pp. 240–248.

[18] R. Guerraoui, M. Herlihy, and B. Pochon, “Polymorphic contention management,” in *Proceedings of the 19th International Symposium on Distributed Computing*. LNCS, Springer, 2005, pp. 26–29.

[19] D. Dice, O. Shalev, and N. Shavit, “Transactional locking II,” in *Proceedings of the 20th International Symposium on Distributed Computing*, September 2006.

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

[21] 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

[22] 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.

[23] 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.

[24] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, “The SPLASH-2 programs: Characterization and methodological considerations,” in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995, pp. 24–36.

[25] G. Reinelt, “TSPLIB95,” Institut für Angewandte Mathematik, Universität Heidelberg, Tech. Rep., 1995.

[26] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III, “Software transactional memory for dynamic-sized data structures,” in *Proceedings of the 22nd annual symposium on Principles of Distributed Computing*. New York, NY, USA: ACM, 2003, pp. 92–101.

[27] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg, “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*. New York, NY, USA: ACM, 2006, pp. 187–197.

[28] 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.

[29] 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

[30] 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.

[31] R. Guerraoui and M. Kapalka, “On obstruction-free transactions,” in *20th ACM Symposium on Parallelism in Algorithms and Architectures*, 2008.

[32] R. Ennals, “Software transactional memory should not be obstruction-free,” Intel Corporation, Tech. Rep., 2006.