Technical Report OUCS-2010-04

View-Oriented Transactional Memory

Authors:

K. Leung and Z. Huang

Department of Computer Science, University of Otago, New Zealand

Status:

Under Submission

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 a View-Oriented Transactional Memory (VOTM) model to seamlessly integrate different concurrency control methods including locking mechanism and transactional memory. The model allows programmers to partition the shared memory into "views" which are non-overlapping sets of shared data objects. A Restricted Admission Control (RAC) scheme is proposed to control the number of processes accessing each view in order to reduce the number of aborts of transactions. The RAC scheme has the merits of both the locking mechanism and the transactional memory. Experimental results demonstrate that VOTM outperforms traditional transactional memory models such as TinySTM by up to five times. Also VOTM outperforms pure lock-based models in applications with long critical sections and has comparable performance with lock-based models in other cases.**

***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]. However, the essential difference behind these two types of models is how concurrency control is implemented when shared data is accessed.

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 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 a novel View-Oriented Transactional Memory (VOTM) 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

[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.

of memory space. Each view can be dynamically created, merged, and destroyed. The most important property for views is that they do not intersect with each other. Before a view is accessed (read or written), it must be acquired; after the access of a view, it must be released. This data-centric model [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 a 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 can be 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. Additionally, RAC can flexibly adjust Q at runtime in order to achieve optimal performance, which will be described in details in Section II-B.

In VOTM, views are classified into three types: Atomic View (AV), Single Writer View (SWV), and Transactional Memory View (TMV). AV is usually declared for primitive variables such as *int* and *long* which atomic operations are implemented with *compare-and-swap* (CAS). AV can be regarded as a special implementation of transactional memory for primitive variables. SWV is for complex data structures and requires exclusive access. TMV is for more complex data structures and allows concurrent access but resolves access conflicts by using transactions. TMV behaves the same as traditional transactional memory and transactions are rolled back if there are conflicts on the access of the same TMV. The transaction granularity for conflict detection for TMV is word-based.

In addition to the transactional mechanism, the RAC scheme is implemented for TMV to dynamically control admission. Comparing to SWV, TMV is more generic than SWV. SWV can be regarded as a special case of TMV whose admission quota Q is set to 1. When a TMV whose Q is greater than 1 is acquired, a transaction is started as in transactional memory; when the TMV is released, the transaction is committed.

A. Contributions of this paper

First, we propose the novel View-Oriented Transactional Memory (VOTM) that seamlessly integrates the merits of both the lock-based and the TM-based approaches and a novel Restricted Admission Control (RAC) scheme that adapts flexibly to runtime contention situations in order to achieve optimal performance.

Second, VOTM ushers in a new programming paradigm, 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 TMVs, but efficient access to them is achieved through RAC.

Third, we implement a VOTM model, called Cocktail. Our experimental results show that VOTM outperforms both purely lock-based systems and TM-based systems while offering the ease of programmability of TM.

The rest of the paper is organized as follows: Section II will discuss the details of the VOTM model and its implementation; Section III will cover experimental results and performance evaluation; Section IV will discuss related work and Section V is about the conclusions and future work.

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 TMVs, so that concurrency control on each TMV 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 TMV, and a hash table can be put into another TMV if they are not required to be accessed atomically 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 TMVs, their accesses are separately optimized, which cannot be achieved by traditional transactional memory.

A. Programming interface

As mentioned before, VOTM provides three types of views: Atomic View (AV), Single Writer View (SWV), and Transactional Memory View (TMV). Both AV and SWV are treated as special cases of TMV. In our current implementation, AV is statically declared, while SWV and TMV are dynamically created.

AV consists of a primitive variable such as *int* and *long*. Its operations such as addition and subtraction are provided by the system. Their atomicity is achieved through instructions such as CAS (compare-and-swap). Its implementation is similar to the roll-back principle of transactional memory, but is more efficient than transactions due to the hardware support of the CAS instruction.

SWV is used for complex data structures that require exclusive access but have a short computation involved. For example, a shared task queue could be put into an SWV if it tends to have high access contention. These data structures are not suitable for optimistic concurrency control approaches such as transactions. As we mentioned before, SWV is a special case of TMV whose admission quota is set to 1.

TMV is suitable for very complex data structures such as graph and hash table. These data structures are often sparsely accessed and involve a long computation. Therefore, they are suitable for optimistic concurrency control approaches. Coarse-grained locking often suffers from low performance on these data structures, but fine-grained locking is very tedious and error-prone for programming on them.

TMV behaves the same as transactional memory, except it has a view identifier used by the programmer to inform the system which group of data objects are going to be accessed.

Figure 1 shows a C example to explain how to use VOTM to make a linked list program.

In the example, *create_view()* creates a TMV for the linked list, and *malloc_block()* allocates a memory block from the TMV. With the creation of the TMV, we allow all nodes in the linked list to be allocated in contiguous memory space. This arrangement currently has no performance advantage in VOTM, but will enable the compiler to detect view accesses easily. We will address possible compiler support in Section V.

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 TMV of the linked list. Compared with the sequential version of the

```
typedef 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, TMV, size);
  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

```
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;
    release_view(vid);
    return;
  }

  /* 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

code snippet, the only extra code is the view primitives, *acquire_view()* and *release_view()*, that demarcate view access.

Deadlock is not possible in VOTM, since all views are prohibited from nested acquisition with *acquire_view()*. If two views need to be acquired in a nested way, they should be either put into the same view initially or merged together dynamically. Fortunately, using TMV, if TMVs 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 TMV.

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

B. Restricted Admission Control (RAC) scheme

We implement the RAC scheme for every TMV. Each TMV consists of memory blocks that may store an entire linked list, tree or graph. Each TMV has an admission quota

Table I
VOTM API

int create_view(int vid, vid_type_t type, size_t size)	Creates a view for *vid*. If *vid* is less than 0, the view ID will be automatically allocated and returned. *type* can be one of: AV, SWV, TMV
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 *ptr* points to that is owned by a view *vid*.
void free_view(int vid)	Frees 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*.

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; if Q is greater than 1, *acquire_view* will either start a new transaction or wait according to the following RAC scheme.

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

- Compare P with Q. If P is smaller than Q, start a new transaction, increase P by 1, and return with success.
- If P equals Q, the calling process is blocked until P becomes smaller than Q.

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

- Try to commit the transaction. If the commit fails, abort the transaction and roll back to restart the transaction.
- 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 TMV is initialized as the maximum number of processes (*NPROCS*). RAC regularly checks the contention situation of the TMV. The contention situation is indicated by the number of aborts as well as the number of successfully committed transactions that are related to the TMV. 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 TMV by halving the admission quota Q. Then, the number of aborts and the number of successful transactions will be reset in the TMV. This process can be repeated until Q reaches 1, in which case the concurrency control is switched to the lock-based approach. 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, if the abort/success ratio is smaller than MIN (currently set to $1/8$), the TMV is considered as having low contention. RAC will increase concurrency by doubling Q. When Q is changed, the numbers of aborts and successful transactions of the TMV will be reset.

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 transaction 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 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.

In our implementation, TinySTM is configured as a redo-log-based TM system. Reading and writing of shared variables are recorded in tentative read- and write-set respectively.

The algorithm of TinySTM is based on the lazy snapshot algorithm (LSA) [15], using the encounter-time locking policy. TinySTM optimally aborts potentially-conflicting transactions early to reduce wasted computation. In the TinySTM algorithm, each memory location is associated with a lock, which has a version number attached. At the beginning of a transaction, the global version clock (GV) is sampled and recorded as the "read version" (RV) of the transaction. When a location is written, if the location is locked by a different process, the transaction is aborted and restarted. Otherwise the lock for the location is acquired and the update is recorded into the write-set.

When a location is read, a transaction must verify that the lock covering the location is not owned or updated by other processes. The transaction reads the lock, then read the memory location and the lock again. If the lock is owned by another process, the transaction aborts. If the version of the lock changes, the read procedure is redone.

Once the location is read, it is checked if it can be used to construct a consistent snapshot. Like LSA, if the version is more recent than the current validity range of the transaction snapshot, it will be "extended" by verifying that every address covered by the read-set is valid and not locked. If the extension is successful, the validity range will be extended to cover the location read; otherwise the transaction will be aborted.

At commit operation, if no conflicts are found, the commit is deemed successful. GV will be atomically incremented by one. Variables in the write-set are written to the actual location. The released version of all the locks in the write-set is set to the current GV.

Each TMV in VOTM is essentially a small TinySTM system, and access to each TMV can be controlled independently so that a TMV with high contention will not affect concurrency of other TMVs which may have low contention. Experimental results in the next section demonstrate that using multiple TMVs 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, 17].

TinySTM is chosen for our implementation because it is a well-maintained, efficient modern TM system [14, 18] with features such as encounter-time locking to reduce time wasted on failed transactions and configurable to be a redo-log system to allow good performance in high abort cases, which makes TinySTM very competitive to VOTM in terms of performance.

Since TinySTM uses 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-A3.

The origin of performance gain in VOTM is very different from traditional TM systems that uses either in-transaction conflict resolution algorithms and/or transaction scheduling algorithms. In-transaction conflict resolution algorithms [19–21] 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 [22–24] prevent conflicts by serializing transactions or limiting 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 TMVs where shared objects that will be accessed together in a transaction are grouped into the same TMV. In this way, restricting access to a TMV with high contention does not affect access to a TMV with low contention, which enables more concurrency. In VOTM, RAC is used as the transactional scheduling algorithm for each TMV, but any in-transaction conflict resolution algorithms can be applied in each TMV. We will further discuss in-transaction conflict resolution and 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 admission to TMVs.

III. Performance evaluation

In this section, we compare the performance of VOTM with the software transactional memory system TinySTM version 1.0.0 [14]. Our benchmark applications include Bayes, Genome, Intruder, Kmeans, Labyrinth, Vacation and Yada from the STAMP transactional memory benchmark suite version 0.9.10 [25] and Travelling Salesman Problem(TSP) from the SPLASH-2 benchmark suite [26]. 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". In each case, speedup is measured against the serial version of the benchmark algorithm. The elapsed time calculated in each case includes initialization and finalization costs. However, the running time of functions that are irrelevant to the original application, such as generation of random input sequences and result-verification, is excluded.

The experimental results are illustrated with speedup curves. For each application, we give the speedup curves using TinySTM and VOTM. In addition, the pure lock-based (i.e. only consists of SWV and AV) version of the application is also implemented as a reference for performance comparison. In cases where the VOTM version does not use TMV, the VOTM version is the same as the pure lock-based version. In the discussion below, N refers to the number of processes, i.e., *NPROCS*.

Speedup is calculated by:

$$speedup = \frac{time_{serial\_version}}{time_{parallel\_version}} \quad (1)$$

To ensure fair comparison, the same serial version of each benchmark application is used as a baseline for calculating speedups of all three systems. Each run is repeated for 10 times and the geometric mean is used.

The rest of this section will demonstrate the three main mechanisms VOTM improves performance:

- Controlling contention by using RAC;
- Providing AV and SWV for shared objects that only need to be held for a short time;
- Partitioning shared objects that are not accessed together atomically into different views.

A. Applications where RAC is a performance enabler

RAC can improve performance of applications that have high contention in part or the entire shared memory yet there is a portion of computation time spent in transactions. RAC achieves good performance by restricting admission to each TMV, therefore reducing conflicts between processes accessing the TMV in high contention situations. The rest of this section will use two applications to show how RAC performs, though it works for other applications as well.

1) Genome: 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 this algorithm, input gene fragments are first inserted in an input fragment hash table in parallel. This step removes duplicated fragments. Then buckets in the input hash table are divided between processes, which calculate the hash function and insert into the array of hash tables. The Rabin-Karp string search algorithm is subsequently used to match gene fragments in parallel. In the VOTM version, a TMV 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.

At $N = 16$, speedup of Genome in TinySTM and VOTM are 5.19 and 6.13 respectively (Figure 3) and number of aborts are 64,595,381 and 572,677 respectively. The 18% improvement of VOTM over TinySTM in Genome can be attributed to restricting access to the shared data structures when contention increases, thus minimizing work wasted by aborted transactions.

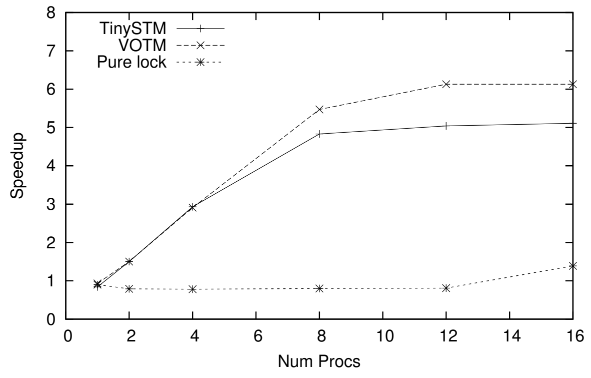

Figure 3. Speedup of Genome

2) Bayes: Bayes has a considerable portion of execution time in transactions that have high contention. Bayes is an application with long transaction and high contention. In this application, shared data structures include a Bayesian graph-like net structure, a task list implemented as a linked list and a few score variables. At the beginning of this algorithm, all processes concurrently build the task list. Then in the main parallel section, each process enters a while loop, where it pops a task from the task list, and then acquires access to the shared net, and calculates changes to the Bayesian net. The access to the shared net is long and has high contention. Then the process may update the global scores and/or insert more tasks to the task list.

Since accesses to the net data structure and the task list are independent, the net is allocated as a TMV and the task list is in another TMV. Contention of the task list and the net can be different, so allocating them in separate TMVs can allow their access to be independently optimized.

To examine the benefit of allocating shared data into different TMVs, an alternative implementation "VOTM 1TMV", which places both the task list and the shared net into the same TMV, is also tested.

Each global score is allocated as an AV and is only accessed when the process does not hold any of the TMVs to avoid unnecessary transaction or lock overheads.

Default parameters "-v32 -r4096 -n10 -p40 -i2 -e8 -s1" are used in this experiment.

From Figure 4, it can be seen that the speedup of both TinySTM and VOTM 2TMV saturates at 3.8 at $N = 4$. However, when N increases further, TinySTM speedup deteriorates because the extra processes can no longer improves performance, but only increases contention and conflicts. At $N = 16$, the speedup of the VOTM 2TMV version stays at 3.84, whereas the speedup of TinySTM drops to 2.19. The speedup of VOTM 1TMV version is 3.58, slightly lower than the 2TMV version, but still 63% better than TinySTM. Therefore, RAC successfully prevents speedup degradation by restricting the number of processes admitted to a TMV. According to our experiment, RAC cuts the number of aborts from 522,972 in TinySTM to 9101 in

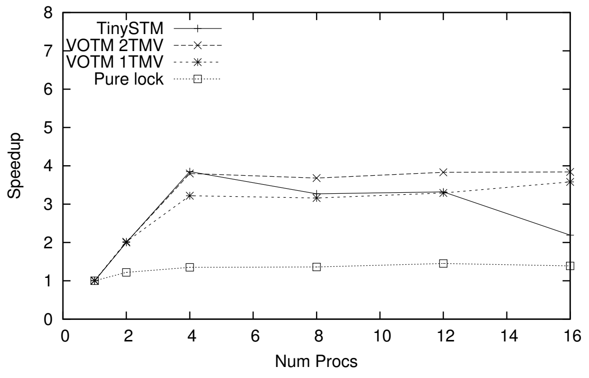

Figure 4. Speedup of Bayes

Table II
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

VOTM 1TMV and 4591 in VOTM 2TMV.

In addition, the performance gain of VOTM 2TMV over VOTM 1TMV can be attributed to separating the net and the task queue into different TMV, so that restricting access to the one structure with high contention will not unduly restrict access to the other structure with lower contention, thus concurrency is maximized.

It is worth noting that, if the net and the task list are accessed exclusively with locks as in the pure lock-based version, speedup vanishes as shown in Figure 4, because most of the computation in this application occur while the data structures are locked.

3) 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 noises, 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 II.

From Table II, 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 TMV becomes 1. Furthermore, the

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

	1(no tx)	1(tx)	2	4	8	16
time(sec.)	27.51	28.34	23.53	12.42	9.4	12.54
No. of aborts	0	0	337	1143	3422	536384

VOTM system allows users to put only deadlock-prone data structures into TMVs and the remaining shared data structures into SWVs which admit single writer but multiple readers.

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 speedup saturates (i.e. the number of processes where maximum concurrency is reached).

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

From Table III, 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 is getting 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.

In Figure 5, we show a scenario to explain 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 TMV.

In Figure 5, 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

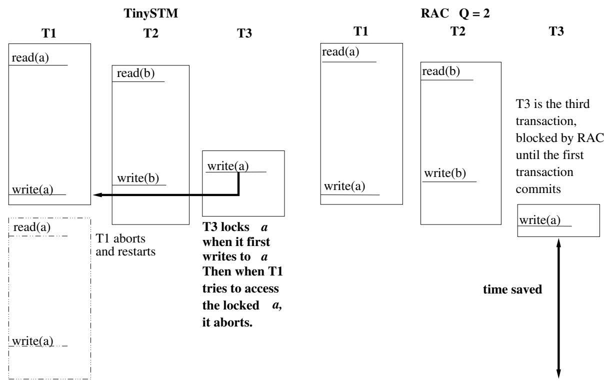
TinySTM
T1
T2
T3
read(a)
read(b)
write(a)
write(b)
write(a)
read(a)
write(a)
T1 aborts and restarts
T3 locks a when it first writes to a Then when T1 tries to access the locked a, it aborts.
RAC Q = 2
T1
T2
T3
read(a)
read(b)
write(b)
write(a)
T3 is the third transaction, blocked by RAC until the first transaction commits
write(a)
time saved

Figure 5. RAC implementation over TinySTM - RAC blocks T3 and prevents it from aborting T1 in high contention

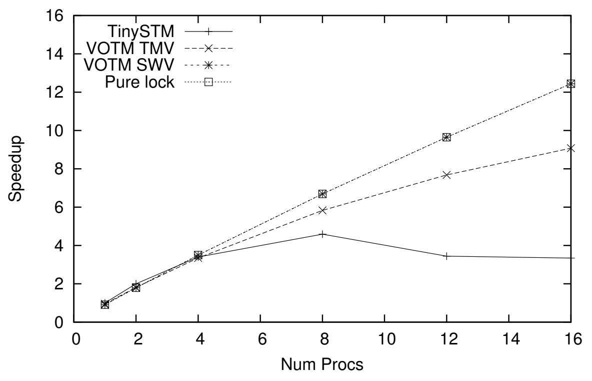

Figure 6. Speedup of TSP

restarts. However, if Q is set to two 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.

In both Bayes and Genome, TMV only allows a certain amount of concurrency to avoid excessive conflicts. After the speedup is saturated, RAC prevents performance degradation by restricting admission to TMVs to the optimum number of processes, whereas traditional TM systems like TinySTM still does not control admission of processes to transactions, and therefore causing excessive conflicts and degraded performance.

B. Applications where performance is improved by using SWV and AV

Lock-based mechanism like SWV is more efficient than transactional memory when the shared object is small but has high contention for accesses. Likewise, atomic operations on word-sized shared variables such as *int* and *long* are better implemented with instructions like CAS than with the heavy-weight TM mechanism.

VOTM can remove unnecessary TM overheads by providing SWV and AV to avoid transactional overhead. One example is the priority queue in the Travelling Salesman Problem (TSP) that is suitable for SWV. Another example is Kmeans where local results are added to a shared array in parallel and each element is better implemented as an AV. As to be shown in our experimental results, Kmeans is five times faster with AV than with transactional memory.

This flexibility shows the strength of VOTM which allows the programmer to improve performance with SWV and AV if appropriate. Since VOTM seamlessly integrates different concurrency control mechanisms through view partitioning, the programmer can flexibly use SWV, AV and TMV to achieve optimal performance within the same programming paradigm.

1) TSP: The Travelling-Salesman Problem (TSP) algorithm is a small-to-medium-size transaction but 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 [27] is used.

In this algorithm, the priority queue (storing partially-evaluated tours) is the shared object. First, the master process pushes the root tour into the priority. Then in a loop, each process pops a tour. If the tour is small, it will be evaluated serially; otherwise, sub-tours will be created and pushed into the priority queue. Accesses to the priority queue are short, but are memory-access intensive, and therefore have high contention.

In the VOTM SWV version, an SWV is allocated for the priority queue, which performs the same as the pure lock-based version. The speedup of VOTM SWV is 12.44, which is four times better than the TinySTM version, as shown in Figure 6.

We have also implemented the VOTM TMV version to show the performance of RAC. In this version, the priority queue is allocated as a TMV with RAC. If the abort/success ratio is above the contention threshold, the TMV dynamically adjusts its admission quota Q and eventually switches to the locking mechanism to stem the aborts. Compared with TinySTM, VOTM TMV improves the speedup to 9.08, which is three times better than TinySTM, as shown in Figure 6.

In the TinySTM version, the *push* and *pop* operations on the priority queue are done in transactions. The high number of aborts (4,150,852,440 at $N = 16$ vs. 15658595 aborts in VOTM TMV) results in the relatively low speedup of 3.04.

In TSP, the time of accessing the priority queue is short but with intense contention, locking mechanism is superior to transaction. The 30% performance loss of VOTM TMV to the VOTM SWV is due to the loss of performance before Q falls back to 1. This loss of performance can be eliminated by simply allocating the priority queue as

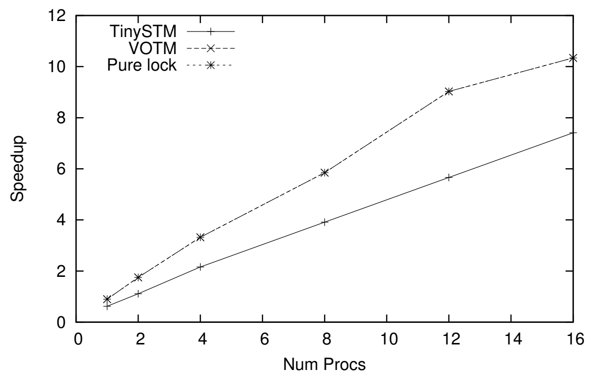

Figure 7. Speedup of Kmeans at low contention

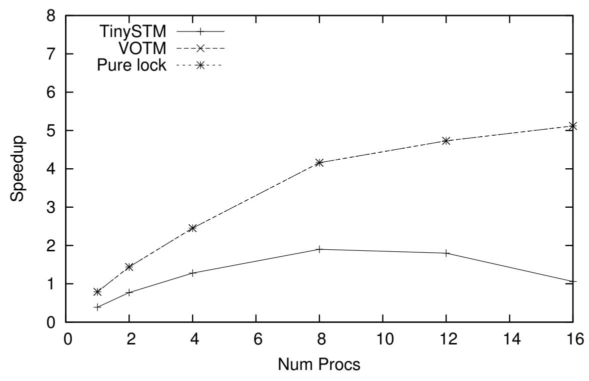

Figure 8. Speedup of Kmeans at high contention

a SWV. This shows the strength of VOTM which allows the programmer to improve performance with SWV if appropriate.

2) Kmeans - incrementing elements in an array: Kmeans is a multi-iteration clustering algorithm, which in each iteration, processes calculated changes independently, and at the end of the iteration, increment each element in the shared array with the local calculated changes. Parameters "-m40 -n40 -t0.00001 -i inputs/random-n65536-d32-c16.txt" and "-m15 -n15 -t0.00001 -i inputs/random-n65536-d32-c16.txt" are used for low contention, and high contention cases respectively. Iterations are repeated until the error falls below the threshold.

In the TinySTM version, updating the entire array is included in a single transaction to avoid the cost for starting/ending a transaction on each element. However in this algorithm, only the increment of each element needs to be atomic, and the arrangement in the TinySTM version causes the read-set and write-set to be unnecessarily large, and therefore increasing the transactional overheads.

VOTM solves this problem by allocating each element in the shared array as an AV, and no TMVs are used. As a result, in both low contention and high contention cases, VOTM has superior performance over TinySTM. The global error value is also allocated as an AV. For $N = 16$, at low contention, VOTM is 50% faster than TinySTM (Figure 7); and at high contention, VOTM is five times faster than TinySTM (Figure 8). In TinySTM, the number of aborts for low and high contention cases are 15,550,412 and 124,546,022 respectively, while VOTM has no aborts due to the use of AV.

The pure lock-based version has the same performance as VOTM since it uses atomic variables which are similar to AV.

C. Applications where performance is improved by partitioning of shared memory into multiple views

Placing shared objects which are not accessed together atomically in different views improves performance, because admission of each TMV is now controlled independently by RAC. Therefore restricting access to a view with high contention does not affect concurrency of accesses to other views with low contention. Bayes and Intruder fit in this class of applications, and Bayes is already discussed in Section III-A2.

1) Intruder: Intruder (from the STAMP benchmark) has short transactions with high contention. In this application, a dictionary (implemented as a self-balancing tree with each node representing a session which consists of a list of packets) is used to store partially-assembled sessions.

The algorithm first puts network packets into a task queue, then at the parallel section, each process inserts a de-queued packet into the correct session in the dictionary and check whether the session of the packet is completely reassembled. If the session is now completely reassembled, it will be checked against known attack signatures.

In the VOTM 1TMV version, all shared data structures are allocated in a TMV. However, in the VOTM 2TMV version, the task queue and the highly contentious dictionary structure are allocated in separate TMVs.

Speedup of both VOTM versions saturates at around 1.59 at $N = 8$ and speedup of the TinySTM version saturates at 1.56 at $N = 4$ (Figure 9). Then as N increases further, speedup of the VOTM 2TMV version decreases slightly to 1.58 at $N = 16$, whereas speedup of the VOTM 1TMV and TinySTM versions drops to 1.11 and 0.52 respectively.

Number of aborts at $N = 16$ for VOTM 2TMV, VOTM 1TMV and TinySTM are 10,396,152, 8,281,400 and 1,238,254,062 respectively.

The performance gain in the VOTM 2TMV version over VOTM 1TMV version can be attributed to allocating the task queue in a separate TMV from the highly contentious dictionary. Therefore access to the task queue is not restricted by RAC access restrictions on the highly contentious dictionary.

In this application, the majority of time is spent in transactions which are memory-intensive and highly contentious. Therefore even when VOTM reduces contention by limiting admission of processes to TMVs, the speedup is still low. For the same reason, the pure lock-based version fails to show speedup because the lock guarding the dictionary

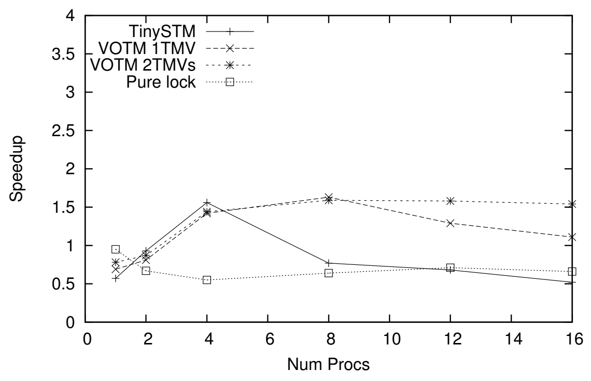

Figure 9. Speedup of Intruder

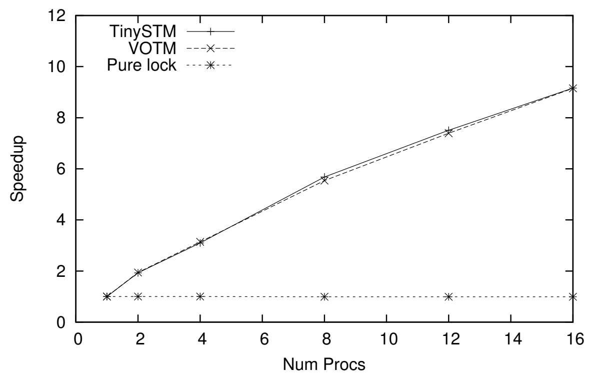

Figure 10. Speedup of Labyrinth

structure abolishes concurrency.

D. Applications where RAC has neutral effects

In applications with long transactions with low contention, such as Labyrinth and Vacation, there are no needs to restrict access, therefore RAC allows free entry to TMV, which is the behaviour of traditional TM models. In this case, RAC cannot improve performance further, however applications show that RAC has little extra overheads.

On the other hand, in applications such as Yada where the majority of computation time is spent in transactions that have high contention, RAC does not improve performance because even if RAC decrease contention and improves progress by restricting admission of processes to the TMV, concurrency will also be abolished in this type of applications.

1) Labyrinth and Vacation - medium and long transaction with low 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 TMV 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 use a lock to protect the access of the grid.

Vacation simulates the operation of a travel agency manager. In this application, the car, room, flight and customer tables are implemented as shared red-black trees. Each process simulates a client. Each task consists of a set of operations including a client adding or removing car, room and/or flight reservations. In this applications, tasks are equally divided between clients (processes). A transaction starts when a task is evaluated. At the end of the task, the process will check whether restraints are met (such as client budget, flight or room occupancy etc.) and will commit the transaction, otherwise the transaction will be aborted and restarted. Transactions are long and with a moderately high memory accesses, but with low contention. Default parameters "-n4 -q60 -u90 -r1048576 -t4194304" are used.

Since all tables (red-black trees) can be accessed in a transaction, they are allocated together in a single TMV in the VOTM version. In addition, since all tables are accessed together atomically, the critical section cannot be broken down in the pure lock-based version, therefore a single lock is used to protect the critical section.

Both applications have low contention. At $N = 16$, Labyrinth has 202 and 195 aborts for TinySTM and VOTM respectively, given that there are 1056 transactions. For Genome, there are approximately 2.48 million transactions, and with 64,595,371 and 83,274 aborts for TinySTM and VOTM respectively. For Vacation, there are approximately 4.2 million transactions, and with 1442 and 1060 aborts for TinySTM and VOTM respectively.

As a result, the RAC protocol in VOTM most of the time admits all processes to the views and thus behaves the same as TinySTM. However, both Labyrinth and Genome show that RAC has little overhead. At $N = 16$, speedup of Labyrinth in TinySTM and VOTM are similar (9.16 and 9.15 respectively) (Figure 10) and

Speedup of Vacation in TinySTM and VOTM are 4.27 and 4.05 respectively (Figure 11). The difference between TinySTM and VOTM in Vacation can be due to RAC overhead in high number of transactions. Despite low contention, the relative low speedup across all cases in Vacation can be attributed to uneven load balancing problems when tasks are divided between processes.

In all applications mentioned above, the pure lock-based version has poor speedup, as every array, every hash table, and in the case of Vacation, the entire shared memory, are protected by a lock and thus accessed serially.

2) Yada - high contention long transaction within which the majority of the computation takes place: Yada implements Ruppert's algorithm for Delaunay mesh refinement [28]. In this application, the mesh is shared between processes and is allocated as a TMV in the VOTM version. The mesh view cannot be subdivided without a complete rewrite of the algorithm. Therefore, the mesh is also protected by a single lock in the pure lock-based version. In this application, transactions are long, and are

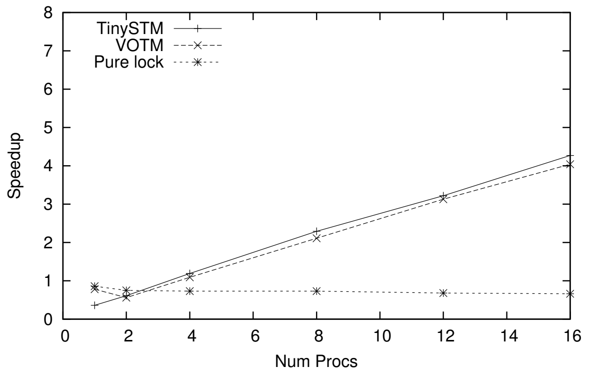

Figure 11. Speedup of Vacation

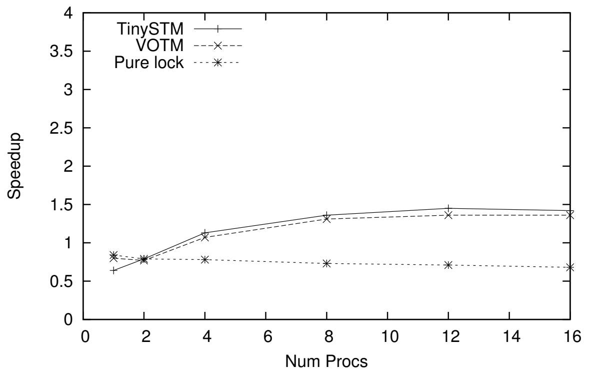

Figure 12. Speedup of Yada

also computational and memory-access intensive. Therefore, they have high contention. Default parameters "-a15 -i inputs/ttimeu1000000.2" are used in this experiment.

In this algorithm, the bulk of the computation in the algorithm occurs in transactions. Therefore as shown in Figure 12, the pure lock-based version (access to the shared mesh structure guarded by a lock, instead of transaction) does not have any speedup. Also despite VOTM cuts the number of aborts from 117,284,245 (TinySTM) to 814754 at $N = 16$, restricting admission still does not help since Yada involves heavy computation in a highly contentious critical section and restricting admission affects its concurrency. In this situation, it is hard for RAC to find the optimal Q (if it exists). If Q is small, the number of aborts is low but there is no enough concurrency; if Q is large, there are sufficient concurrent computational transactions but there are many conflicts as well. Therefore, the adjustment of Q is neutral to the performance. As a result, speedup of VOTM (1.36) is slightly lower than speedup of TinySTM (1.42) due to the extra overhead of RAC. However, this is the only application we find so far that RAC cannot help much under high contention situations.

IV. Related work

There have been a large amount of work on contention management in TM systems in the past years. Contention management mechanisms can be divided into two categories:

A. *In-transaction conflict resolution*

Many existing algorithms on TM contention management resolve conflicts *within* a transaction after these conflicts occur, and decide which transaction proceeds/commits and how other transactions are blocked or aborted [19–21, 29–32].

For example, there has been some recent work on in-transaction conflict resolution published by the Tanger group at the Universität Dresden [32]. It proposes to allocate different disjoint memory structures into different "memory pools". Since a program may have disjoint shared memory structure that have different access pattern (e.g difference in contention and ratio of RO or RW accesses), access to each memory pool can be separately optimized. This optimization shares the same data-partitioning philosophy as VOTM. However, instead of restricting the number of processes accessing the same TMV as in our RAC scheme, this optimization dynamically adjusts the *granularity* of the pool from single-pool to word-size in high contention to avoid conflicts. Since RAC and the adjustment of granularity are complementary, the system in [32] can adopt RAC to further improve performance if the number of processes accessing each pool is tracked.

TM systems like DSTM [19, 29], SXM [20] and McRT-STM [30] also adopt advanced contention management. Like TinySTM, they use an encounter-time locking system that locks each object as it is accessed. When more than one transaction attempts to have write access to the same object, the contention manager in the TM systems will choose one transaction to proceed according to the chosen contention policy, with the aim to minimize wasted computation and ensure progress of all transactions. Other transactions will be blocked or aborted according to the policy. Blocked transactions will retry using a protocol similar to exponential backoff, and can abort if necessary. Compared with the late-locking system used in TL-2 [21] and NOrec [33], the early-locking system can reduce wasted computation by aborting early.

However, in either early-locking or late-locking cases, in-transaction conflict resolution algorithms mentioned above only solve conflicts after they have been detected, but processes are still freely admitted into transactions and/or restart aborted transactions, therefore aborts cannot be stemmed in high contention and work is still waste by transactions that eventually aborts.

B. *Transaction scheduling*

Recently, some transaction scheduling algorithms has 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.

For example, the admission control algorithm in [22] is similar to RAC, where the transaction commit ratio (TCR) (the percentage of committed transactions in the total number of transactions executed) is used to determine contention and adjust admission quota (Q). If TCR is below the lower threshold, contention is considered as high, and Q will be decreased. Conversely, if TCR is above the upper threshold, contention is considered as low, and Q will be is increased (if Q is not already equals to the number of processes). In addition, if Q changes, the TCR sampling interval decreases to improve response time to contention changes. If the admission quota stays over a certain number of samples, then the TCR sampling interval decreases to reduce overheads.

Transaction scheduling algorithms such as [23] use a process-local contention score and when a process experiences high contention, it queues the starting transaction to a central scheduler, which will execute queued transactions serially. [24] 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.

Both transaction scheduling algorithms mentioned above control access to the entire TM as a whole, whereas VOTM allows separate access control to different TMVs. As discussed before, in VOTM, concurrent access of a TMV with low contention will not be affected by restrictions placed on other TMVs with high contention. This separation in VOTM has performance advantage as shown in our experiment, which is not attainable with the above scheduling algorithms. However, in the future, we can introduce the adaptive sampling interval in [22] to further improve its response to contention level changes for each TMV in VOTM.

V. Conclusions and future work

VOTM seamlessly integrates locking mechanism, atomic variables, and transactional memory into one programming paradigm. It can take advantage of the merits of both the pessimistic and the optimistic approaches to concurrency control. Among the three types of views in VOTM, AV allows atomic operations on primitive shared variables with minimal cost. SWV allows single-writer multiple-reader access of complex data structures and is suitable for operations such as I/O that have side-effects. TMV takes the optimistic approach and allows concurrent access to more complex data structures, and relieves contention using the RAC scheme proposed in this paper.

VOTM allows the programmer to optimize performance by placing each shared variable and data structure into the correct type of view according to its access pattern. In cases where the lock-based mechanism performs the best, the programmer can use SWV and AV to enable high performance. In cases where transactional memory is more convenient and performs better, the programmer can choose TMV. With TMV, the programmer does not need to worry about the 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. In future, for cases where TMV performs better, we could even provide API for programmers to fine-tune their programs by setting the admission quota Q of a view to a specific value (e.g. 1 or some optimal value that RAC performs the best).

This new programming paradigm enables transactional memory to interact better with traditional lock-based code, especially from semantic point of view. With VOTM, lock-based code can be easily converted into code with SWV or TMV by replacing *acquire_lock*/*release_lock* with *acquire_view*/*release_view* and defining proper views for the related data. This conversion ensures the consistent view-oriented semantic of the whole program, without worrying about the differences between transactional code and the non-transactional code. In VOTM, the transactional semantic can be clearly confined to a view when the view is acquired. That is, we need only roll back the view and the automatic (in-stack) variables when the transaction aborts.

In this paper, we have proved the concept of VOTM through a plain implementation (version 0.1), though VOTM is a new programming paradigm with promising potentials and interesting issues to be further addressed. The paper demonstrated memory partitioning through views is feasible and efficient, and enables better performance in transactional memory. It showed a nice tradeoff that has brought performance gain as well as the integration of different concurrency control mechanisms. Through partitioning the shared memory, multiple concurrency control mechanisms can be adopted and implemented on per view basis. This leaves much room for optimization of concurrency control on per view basis for future research.

One issue with VOTM is that VOTM programs possibly have the problem of priority inversion [34], since SWV uses the locking mechanism to guard the view as in the lock-based approach. However, if necessary, programmers can choose to use TMV and AV only to avoid such a problem in VOTM programs.

Another issue with VOTM programs is the 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 [35]. Even though this feature is arguably necessary [36], 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 the 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.

A further issue with the current VOTM model is that forbidding nested view acquire can affect program composability. For example, if a library function acquires a TMV, the program would not be able to hold any views while calling the library function. This issue will be addressed in the future by allowing conditional nested view acquisition, so that deadlock will not happen while nested view acquisition is allowed conditionally. For example, we can forbid SWV in library functions.

VOTM offers some optimizations for compiler support of TM. Since data objects in a view are allocated contiguously in memory space, it is easier for the compiler to detect view access and automatically insert *Tx_read* and *Tx_write* as required by software TM. Also enforcement of view access can be imposed by the compiler. If a view is acquired, the compiler can check if the accesses are inside the address range of the view. If a program has acquired one view but access another view, the compiler can notify the programmer with error messages. Currently our VOTM implementation does not have such enforcement. However, the lack of this enforcement does not affect the correctness of the VOTM implementation.

As a future work, we are going to investigate compiler support for VOTM programming model such as detection of view accesses and static declaration of SWV and TMV. We will also investigate semantics and performance of conditional nested view acquisition in VOTM. Other interesting future issues are related to RAC, such as adaptive adjustment of sampling interval and finding the optimal parameters such as *MIN* and *MAX* for the abort/success ratio. We will also investigate how to best adjust the admission quota Q using more benchmarks such as RMS-TM [37] and STMBench [38].

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*, ser. ICPP '08, 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*, ser. ICPP '05. 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*, ser. HPCA-13, 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*, ser. PPoPP '08. 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*, ser. DISC '06, September 2006.

[16] S. Chaudhry, "Rock: A SPARC CMT processor," Sun Microsystems, Tech. Rep., 2008.

[17] 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, ser. ASPLOS '09. ACM, 2009, pp. 157–168.

[18] R. Guerraoui and M. Kapalka, "The semantics of progress in lock-based transactional memory," in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '09. ACM, 2009.

[19] 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*, ser. PODC '05, M. K. Aguilera and J. Aspnes, Eds. ACM, 2005, pp. 240–248.

[20] R. Guerraoui, M. Herlihy, and B. Pochon, "Polymorphic contention management," in *Proceedings of the 19th International Symposium on Distributed Computing*, ser. DISC '05. LNCS, Springer, 2005, pp. 26–29.

[21] D. Dice, O. Shalev, and N. Shavit, "Transactional locking II," in *Proceedings of the 20th International Symposium on Distributed Computing*, ser. DISC '06, September 2006.

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

[23] 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*, ser. SPAA '08. New York, NY, USA: ACM, 2008, pp. 169–178. [Online]. Available: http://doi.acm.org/10.1145/1378533.1378564

[24] 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*, ser. PODC '09. New York, NY, USA: ACM, 2009, pp. 7–16.

[25] 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*, ser. IISWC '08, September 2008.

[26] 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*, ser. ISCA '95, 1995, pp. 24–36.

[27] G. Reinelt, "TSPLIB95," Institut für Angewandte Mathematik, Universität Heidelberg, Tech. Rep., 1995.

[28] M. Kulkarni, L. P. Chew, and K. Pingail, "Using transactions in Delaunay mesh generation," in *Workshop on Transactional Memory Workloads*, 2006.

[29] 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*, ser. PODC '03. New York, NY, USA: ACM, 2003, pp. 92–101.

[30] 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*, ser. PPoPP '06. New York, NY, USA: ACM, 2006, pp. 187–197.

[31] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc, "Practical weak-atomicity semantics for Java STM," in *20th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '08. New York, NY, USA: ACM, 2008, pp. 314–325.

[32] T. Riegel, C. Fetzer, and P. Felber, "Automatic data partitioning in software transactional memories," in *20th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '08, June 2008.

[33] 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*, ser. PPoPP '10. New York, NY, USA: ACM, 2010, pp. 67–78.

[34] B. W. Lampson and D. D. Redell, "Experience with processes and monitors in Mesa," *Commun. ACM*, vol. 23, no. 2, pp. 105–117, 1980.

[35] R. Guerraoui and M. Kapalka, "On obstruction-free transactions," in *20th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '08, 2008.

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

[37] G. Kestor, S. Stipic, O. S. Unsal, A. Cristal, and M. Valero, "RMS-TM: A transactional memory benchmark for recognition, mining and synthesis applications," in *Proceedings of the 4th Workshop on Transactional Computing*, ser. TRANSACT '09, 2009.

[38] R. Guerraoui, M. Kapalka, and J. Vitek, "STMBench7: A Benchmark for Software Transactional Memory," in *Proceedings of the Second European Systems Conference*, ser. EuroSys '07, 2007.