TrC-MC: Decentralized Software Transactional Memory for Multi-Multicore Computers

Kinson Chan, Cho-Li Wang
The University of Hong Kong
{kchan, clwang}@cs.hku.hk

***Abstract*—To achieve single-lock atomicity in software transactional memory systems, the commit procedure often goes through a common clock variable. When there are frequent transactional commits, clock sharing becomes inefficient. Tremendous cache contention takes place between the processors and the computing throughput no longer scales with processor count. Therefore, traditional transactional memories are unable to accelerate applications with frequent commits regardless of thread count. While systems with decentralized data structures have better performance on these applications, we argue they are incomplete as they create much more aborts than traditional transactional systems. In this paper we apply two design changes, namely zone partitioning and timestamp extension, to optimize an existing decentralized algorithm. We prove the correctness and evaluate some benchmark programs with frequent transactional commits. We find it as much as several times faster than the state-of-the-art software transactional memory system. We have also reduced the abort rate of the system to an acceptable level.**

I. Introduction

Parallel programming has become an important issue since the beginning of this century. With the appearance of modern multicore processors, domestic computers are able to run multiple threads concurrently. It becomes a necessity to write parallel programs in order to fully utilize the computation power. The old lock-based programming is no longer enough as it encourages excessive mutual exclusion, and is error-prone (e.g., deadlock) while lacking simple error detection mechanisms.

Software transactional memory (STM) [1] is a promising next-generation programming paradigm. Critical regions are replaced by transactions, each of which is executed atomically, consistently, and also in isolated manner. It promises easy programming [2] similar to coarse-grain locking but the programs remain scalable. There are lots of researches through the last decade, including object-based [1], [3], [4] and word-based [5], [6], [7], [8] implementations. As the systems do not require programmers to specify locks or labels on the transactions, they usually follow a single-lock atomicity model. Intuitively, *clock*, a common meta variable, is required to serialize the transactions in a globally known order. Before a transaction commits and makes its effect visible in the main shared memory space, the clock is usually updated once.

Although multicore processor accelerates meta-data sharing though the common cache, there are situations clock value sharing is still slow. For example, when there are multiple multicore processors on a system, data sharing between two threads on two processor packages is as slow as before, as we will further discuss in the next section. The clock, which updates frequently, becomes a serious data sharing hotspot. To update clock values in commit procedure, the processors have to exchange large amount of cache coherence messages through the long circuit on motherboard or processor modules. Much less time is available for useful computations. To distinguish from other types of contentions, in the following context, we call this particular phenomenon as *clock contention*.

To actually claim STM is truly ready for *general purpose* programming, besides assuring it works for long transactions [9], we should also ensure it works when there are frequent transactional commits. Unfortunately, the clock contention makes traditional STM unable to accelerate this type of applications.

There are many attempts to avoid having the clock contention but the results are not satisfactory because they usually increase the likelihood of false conflicts. In this paper we start from an existing distributed-clock STM design and make changes to reduce the false conflicts. Results show distributed-clock protocol outperforms the traditional counterpart in particular system workload. The contributions of this paper are as follows:

- We identify a current (rather than future) computing hardware bottleneck, which obstructs STM from scaling in some transactional workloads.
- We extend a distributed-clock STM with two optimization changes, and prove the correctness of the resultant design.
- We evaluate the distributed-clock STM, including our optimizations, with some real world benchmark programs, and recorded huge performance difference.
- We successfully reduced the abort rate of the distributed-clock STM to an acceptable level.

II. Background

Clock-based validation is a necessity to avoid the quadratic-time validation complexity. Meanwhile clock sharing hits the wall of cache coherence capacity. Although distributed clock design works around the clock contention issue, it has higher storage overhead and more false conflicts.

A. Clock-based validation in STM

The very early STM do not have a common clock. In DSTM [3], there is a meta-object pointing to different versions of a transactional object. A consistent snapshot is essential to avoid accidental infinite loops or invalid array references.

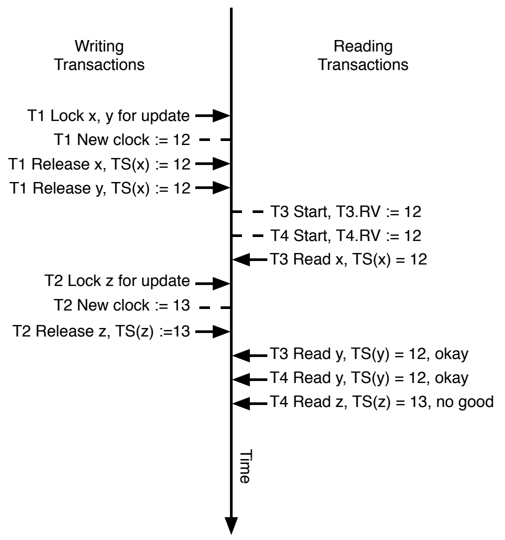

Fig. 1. Snapshot validation in TL2

Therefore, for every new open-for-read operation, all the previous reads have to be verified again. When a transaction opens n objects for reading, it entails checking the meta-objects for $1 + 2 + \cdots + n = O(n^2)$ times.

Introduction of clock-based validation [10] makes the quadratic validation time complexity unnecessary. In the global shared memory, there is a common clock which is incremented every time a transaction commits. Objects (or data words) are given corresponding timestamps, which indicate the relative time (in clock values) they were last updated. An example is given on Figure 1. When a transaction starts on thread i, it copies the clock value into its thread local variable $T_i.RV$. Through execution, it ensures the timestamps that it encounters must have a value not bigger than $T_i.RV$. This way, all the data that the transaction reads belong to a single consistent snapshot *when* the clock value is copied at beginning. (e.g., T_3 in the figure)

There are two drawbacks in this clock-based design. Firstly, the constant $T_i.RV$ makes the transaction unable to handle any data that are updated after the transactional start, even if it still forms a consistent snapshot. (e.g., T_4 in the figure) Secondly, the common clock mandates commit procedure to visit a common memory spot. Therefore, the processors' cache coherence capacity limits the transactional commit rate. The first issue is solved by timestamp extension technique [11]. We investigate on coherence capacity in the next subsection.

B. Cache coherence on modern hardware

Modern computers are equipped with multicore processors (CMP). A multicore processor features several computation cores and a shared cache. With chip multithreading (CMT) technology, a core can run multiple threads concurrently. Threads within the same core share the L1 cache while threads on different cores (but same processor) share L2 or L3 cache. The cache sharing makes data sharing much faster, as the data remains on cache instead of being copied to the main memory modules.

TABLE I
Ping-Pong Rates (Round trips per second) of some Intel Multicore Processors

Processor	Clockspeed	Intra-Die	Inter-Die
Core 2 Quad Q6600	2.40 GHz	1.6×10^7	3.9×10^6
2× Xeon E5540	2.53 GHz	1.3×10^7	5.4×10^6
2× Xeon E5550	2.66 GHz	1.5×10^7	6.3×10^6
2× Xeon X5670	2.93 GHz	1.4×10^7	6.2×10^6
4× Xeon X7750	2.00 GHz	5.4×10^6	9.0×10^5

To pursue high performance, high-end computers are equipped with multiple multicore processors. This complicates the cache-sharing situation. Threads on different processors do not share any common cache. To share data, the cores exchange cache coherence signals through the long circuit on the motherboard or processor modules. In this case, data sharing is almost as slow as before.

We run ping-pong tests on several systems equipped with typical multicore processors. Our aim is to evaluate how fast data sharing can be. Performance data on inter-thread sharing may give us some inspiration how STM policies and protocols can be designed. Results are shown on Table I. For example, a pair of Xeon E5540 processors can exchange cache ownership for 5.4×10^6 round trips in a second, or 1.1×10^7 transfers only. If there is a common clock in STM and transactions commit evenly on both processors, this is likely the limit for the transactional throughput. The actual limit may be even lower, given there are more timestamps or shared data cache invalidations to handle. For example, with a common clock, we can only achieve throughput of at most 5.8×10^6 transactions per second with *ssca2* benchmark on two E5540.

C. Distributed Clock STM

To work around the clock contention issue, various solutions are suggested. Avni proposes TL2C [12], which provides a distinct clock per thread, thus removing the memory hotspot in commit procedure. In the proposal, a timestamp is a tuple of the thread id and the clock of the last writer. On each thread i, there is a thread local counter $T_i.TLClock$, as well as a cache storing $n - 1$ clock values from different threads, $T_i.CArray[0 \cdots n]$. [1]

When thread i updates some value and commits a transaction, it increments its clock $T_i.TLClock$ and set each of the related timestamps to be a tuple of its thread id and the new clock value. When transactional thread j performs a read operation, it encounters a timestamp with thread id i and clock value t. If $i = j$, there is no need for validation; otherwise the transaction considers the read as valid only if $T_j.CArray[i]$ is not smaller than t. If this condition is not satisfied, the transaction updates $T_j.CArray[i]$ to be t, aborts the speculative computation, and starts from transaction begin again. Note the threads do not visit other threads' $TLClock$

[1] Although there are n array elements, $T_i.CArray[i]$ is not used as a thread never caches its own clock.

counters. They use the timestamp table as the only indirect way to acknowledge newer clock numbers. Although it causes extra unnecessary aborts, it avoids clock sharing and keeps the $T_i.TLClock$ of each thread i resides exclusively within the respective L1 caches.

D. Disadvantages of Distributed Clock Systems

Although this solution makes clock contention less frequent, the inventors also notice the design introduces more chance for a transaction to abort. We suspect it is because the updated clocks cannot propagate to new transactions as fast as before, thus the numbers in $T_i.CArray$ are smaller than optimal. When there are more threads, there are also more clocks in $T_i.CArray$ to keep up-to-date, implying more aborts are necessary. Storage overhead may also become a design problem in future. A timestamp is usually as long as a machine word (64 bits) only, which is now responsible storing both a thread id and a clock. When there are more threads, there will be more bits taken from the latter for the former. Clock overflow issue will appear when there are too few bits for the clock. Also, in a system with n threads, each thread has an array $CArray[0 \cdots n]$, resulting $O(n^2)$ storage overhead.

We believe eliminating some of the clocks wisely improves the system performance. Timestamp extension technique may also help updating the cached clocks in a live transaction, and thus reducing the abort ratio. Meanwhile, it remains an open question whether these can be combined with the distributed clock design. To our knowledge, there have been no attempts in doing so.

III. Related Works

There are a few solutions targeting to relieve from global clock contention or remove the global clock. An incomplete list is as follows.

The early-time transactional memory systems do not have any common clocks. Instead, there is a status field on each meta object. Disjoint transactions do not overlap in commit data path, and can commit concurrently. Unfortunately, to have a consistent memory view in a transaction, these systems need to revalidate all previous reads per new read operation, resulting quadratic time complexity.

TL2 [5] is a state-of-the-art word-based STM. According to Avni [13] and Hill [14], there are several global versioning techniques with TL2. Among them, GV5 defers updating the global clock as late as another transaction encounters a false conflict due to the stale clock. This makes global clock updates less frequent and therefore relieves the clock contention issue. As the stale clock values also create more aborts, there is another scheme, GV6, updating the clock sparsely with a given probability. These schemes have not removed the common global clock, and is also generating more false conflicts then the original GV4 design.

TL2C [12] is another variant of TL2, with distributed global clock variable. Avni also provided a TCV model [13] to prove the correctness of the TL2 family including TL2C. Unfortunately TL2C also suffers from the increased abort ratio like other TL2 siblings. While our solution is similar to TL2C, we avoided large portion of the aborts by applying multicore-friendly partitioning, as well as reusing the timestamp extension from TinySTM [7].

TLRW [15] abolishes the global clock completely, only relying on the vector timestamp table for disjoint transactional executions. To avoid the quadratic validation overhead, the transactional reads are semi-visible in form of reader counters. A transaction cannot start writing on a shared data until there are not any readers or writers. This design eliminates the need for any validations. Meanwhile, as readers also have to modify the counters, there are cache invalidations among the processors even when all the transactions are read only. Attiya et al. show [16] that to ensure successful commits of read-only transactions, there must be $\Omega(t)$ memory updates on metadata for the t read operations.

There are several attempts trying to remove the global clock or the timestamp mechanism, namely TML [17], NOrec [18], RingSTM [19], InvalSTM [20], etc. While these designs successfully removed the bulky timestamp table, they retain some global data structure, such as a single version number in TML and a ring in RingSTM. Transactions have to compete updating the global data structure in order to serialize among themselves, which means the contention issue is not ever solved.

There are also several distributed transactional memory systems which are meant to span over a number of cluster nodes in computation. They require either a broadcast [21], [22], [23] or a token server [24], [25] in the commit procedure, with the goal to serialize the transactions. The data contention, in addition to the network latency, make these systems not ready for frequent transactions.

There are also several works that do not follow the intuitive single-lock consistency, such as z-linearization [26] and snapshot isolation [27]. It is natural these systems touch less common data structure in commit procedure, but their programming models are not intuitive to existing programmers who used to use locks.

Lastly, there are solutions [28] that require manual partitioning of data into multiple instances of transactional memory systems. These solutions require heavier programming effort, making them useful in large cluster environments but not among simple programmers.

IV. Design Changes to Distributed Global Clock Algorithms

We propose changes to the distributed clock algorithm TL2C. We first define several data structures in Table II and show the resultant algorithm on Table III and IV. We call the design as TrC-MC, standing for "Transactional Consistency for Multicore". For simplicity, we assume each data word is associated with an unique timestamp and a word is never written for the second time within a single transaction. [2]

[2]In actual implementation, we have handled these conditions. But they are beyond the scope of this paper.

TABLE II
TRC-MC VARIABLES

Variable	Usage	Scope
j	zone number found on a timestamp	function
t	clock number found on a timestamp	function
x	shared memory to be accessed	function
v	data to be written on shared memory	function
i	current thread number	thread
k	zone number of current thread	thread
T_x	transaction descriptor for thread x (T_i stands for current thread's tx descriptor)	thread
$T_x.CArray$	cache clock array for thread x	thread
$T_x.readset$	read set of T_x, containing tuples of locations and timestamps	thread
$T_x.writeset$	write set of T_x, containing tuples of locations, values and timestamps	thread
Z_x	zone descriptor for zone x (Z_k stands for current zone's descriptor)	zone
$Z_x.CArray$	cache clock array for zone x	zone
$Z_x.TLClock$	actual clock for zone x	zone

TABLE III
TRC-MC READ WRITE PROTOCOL

```
1   function stm_read($x$)
2       if $x \in T_i.writeset$
3           return $T_i.writeset[x]$
4       end if
5       $TS2 \leftarrow$ timestamp($x$)
6       $value \leftarrow x$
7       $TS \leftarrow$ timestamp($x$)
8       if is_locked($TS$)
9           stm_abort( )
10      else if $TS2 \neq TS$
11          restart this function
12      end if
13      $j \leftarrow$ zone_bits($TS$)
14      $t \leftarrow$ clock_bits($TS$)
15      if $t > T_i.CArray[j]$
16          share_clock($j$, $t$)
17          if (extend( ) = success)
18              $T_i.CArray[j] \leftarrow t$
19              restart this function
20          else
21              stm_abort( )
22          end if
23      end if
24      $T_i.readset \leftarrow T_i.readset \cup (x, TS)$
25      return $value$
26  end
27
28  function stm_write($x$, $v$)
29      $TS \leftarrow$ timestamp($x$)
30      if is_locked($TS$)
31          stm_abort( )
32      else if $TS \neq$ cas(timestamp($x$), $TS$, $my\_lock$)
33          restart this function
34      end if
35      $j \leftarrow$ zone_bits($TS$)
36      $t \leftarrow$ clock_bits($TS$)
37      if $t > T_i.CArray[j]$
38          share_clock($j$, $t$)
39          timestamp(x) $\leftarrow TS$
40          if (extend( ) = success)
41              $T_i.CArray[j] \leftarrow t$
42              restart this function
43          else
44              stm_abort( )
45          end if
46      end if
47      $T_i.writeset \leftarrow T_i.writeset \cup$ ($x$, $v$, $TS$)
48  end
```

TABLE IV
TRC-MC AUXILIARY PROTOCOL

```
49  function stm_begin( )
50      for each zone $j$
51          $T_i.CArray[j] \leftarrow Z_k.CArray[j]$
52      end for
53  end
54
55  function stm_abort( )
56      for each ($x$, $v$, $TS$) in $T_i.writeset$
57          timestamp($x$) $\leftarrow TS$
58      end for
59      restart from transaction begin
60  end
61
62  function stm_commit( )
63      if extend( ) $=$ failure
64          stm_abort( )
65      end if
66      $wv \leftarrow Z_k.TLClock + 1$
67      for each ($x$, $v$, $TS$) in $T_i.writeset$
68          $x \leftarrow v$
69          timestamp($x$) $\leftarrow$ ($i$, $wv$)
70      end for
71      share_clock($k$, $wv$)
72  end
73
74  function share_clock($j$, $t$)
75      $Z_k.CArray[j] \leftarrow$ max($Z_k.CArray[j]$, $t$)
76  end
77
78  function extend( )
79      for each ($x$, $TS$) in $T_i.readset$
80          if timestamp($x$) $\neq TS$
81              return failure
82          end if
83      end for
84      return success
85  end
```

A. Zones

In the original TL2C system, all the STM metadata, except timestamp table, are decentralized to be thread local variables. In our proposed system, we have a new intermediate layer of information sharing called *zones*. Each of the threads (or transactions) is assigned to belong to a zone. Ideally, threads running on the same processor are assigned to the same zone. Alternative configurations mimic other STM. For example, when all threads are assigned to a single zone, it is similar to a traditional STM with a single clock. When each of the threads is assigned a unique zone, it is similar to the TL2C design.

While a thread still possesses some thread locals such as read set and write set, threads within a zone also share the same clock and cache clock array. In the commit procedure, a thread updates the clock in the zone (line 71), allowing new threads within the zone acknowledge the new clock without extra effort. When a thread encounters a new timestamp clock from a foreign zone (line 15, 37), it shares with other zone members by posting it onto the zone data structure (line 16, 38). Other zone members receive the new cache clocks when they start the upcoming transactions (line 49–53). In this design threads see new clocks faster than the original decentralized design, yet do not incur much contention as threads within a zone share a common cache.

B. Timestamp Extension

It is inevitable to encounter newer timestamp clocks in TL2C because there are not other means to acknowledge the new values for the $CArray$. Unfortunately encountering a new timestamp value also implies an abort in the design.

We reuse the design of timestamp extension and apply to TrC-MC. The *extend* function is called every time a new timestamp clock is encountered (line 17, 40). It checks the content in read set again, asserting all the originally-obtained timestamp values are still valid (line 79–83). If the extension fails, the transaction aborts as usual.

V. Correctness Proof

In this section we prove correctness of zone partitioning with mathematical induction. We first let $|Z|$ be number of zones. To prove the correctness of our new design changes, we first assume the TL2 algorithm is correct. Readers may find proofs on the original TL2 paper [5] and the TCV model [13]. Although there is a proof for TL2C already, we believe the addition of zone concept makes a separate proof necessary.

Lemma 5.1: A transaction T_1 is consistent if memory referred by its readset is not modified, or modified by T_2 while T_1 has not yet encountered any data updated by T_2 and subsequent transactions.

Proof: If data referred by a readset is not modified, it is trivially true. If it is partially modified, without actually reading anything written by T_2 and subsequent transactions, the snapshot is just the same as if T_2 did not exist. ∎

Lemma 5.2: All the zone-shared $Z_k.CArray[j]$ cache clocks are under-estimations of the respective true clocks $Z_j.TLClock$.

Proof: The $Z_j.TLClock$ is already incremented in *stm_-commit* procedure before the value is written onto the timestamp table (line 66, 69). Contents in $Z_k.CArray$ can be only be updated through the *share_clock* function, which is invoked only when the new timestamp clock is seen on timestamp table (line 15, 37). Therefore, $Z_k.CArray[j]$ cannot hold any clock values higher than the true clocks $Z_j.TLClock$. ∎

Lemma 5.3: The thread local $T_i.CArray[j]$ are also under-estimations of the true clocks $Z_j.TLClock$.

Proof: $T_i.CArray$ is a snapshot of $Z_k.CArray$ when a transaction starts at *stm_begin* function. Therefore it is also an under-estimation of $Z_k.CArray$, and thus the true clocks $Z_j.TLClock$. ∎

When $|Z| = 1$, the system is just the same as TL2 after partitioning, and therefore correct: There is a single clock $Z_1.TLClock$, shared by all the threads. Although there is also a cache clock $Z_1.CArray[1]$, it represents under-estimation of the actual clock, and creates false-conflicts but not invalid commits.

Assume the system is correct when $|Z| = n \geq 1$, with zones $Z_{1\cdots n}$. Consider the case when $|Z| = n + 1$, that is, with the extra zone Z_{n+1}. We consider a case T_1 has read a shared variable X and T_2 has subsequently overwritten X with a new value. As the zone numbers are dummy, without loss of generality, we assume T_1 is within $Z_{1\cdots n}$ and T_2 is within Z_{n+1}. To have the system considered working correctly, T_1 must follow Lemma 5.1, and detect its consistent snapshot with X is broken when it encounters any data location Y that is also updated by T_2.

T_1 was able to start transaction and read X before T_2 commits a timestamp operation on X. Through Lemma 5.3, we know the $T_1.CArray[n + 1]$ is an under-estimation of $Z_{n+1}.TLClock$ right before T_2 commits, and is strictly smaller than what T_2 writes onto the timestamp entries. When T_1 reads any timestamp written by T_2, the *extend* function is called and it detects timestamp of X is being changed.

If another transaction, T_3, modifies Y, erasing the timestamp trace of T_2, T_1 is still able to detect the change. If T_3 belongs to $Z_{1\cdots n}$, changes to X must be detected as *extend* function is invoked. (We assume the system is correct among $Z_{1\cdots n}$.) If T_3 belongs to Z_{n+1}, the detection is just the same as detecting T_2, as the $Z_{n+1}.TLClock$ is further increasing. Therefore we know given the system is correct when $|Z| = n$, the system is also correct when $|Z| = n+1$. By mathematical induction, we now know the system is valid for all positive natural numbers of zones. ∎

With the basic zoned timestamp mechanism proved, we now discuss about timestamp extension. In order to update any entries in $T_i.CArray$ (line 18, 40), the *extend* function is called once. First of all, we know write set entries are already locked and are not possible to be modified. Next, the *extend* function revisits all the read set entries, ensuring the timestamp is the same as before. If the *extend* function returns success, the thread has revisited all of the read set and realized there are not any changes, which is effectively the same as restarting the transaction and finding the execution result is the same. Therefore, $T_i.CArray$ can be updated to any valid contents as long as Lemma 5.2 and 5.3 are followed. ∎

VI. Evaluation

A. Test Platforms and Subjects

We took experiments on two computers—Dell PowerEdge M710 and PowerEdge M910. They are respectively equipped with two Xeon E5540 (2.53 GHz) and four Xeon X7550 (2.00 GHz). With hyper-threading (CMT) technology enabled, they are able to run 16 and 64 concurrent threads respectively. The computers run Fedora 11 and CentOS 5.4 respectively. We use the same GCC compiler version 4.4.1 for compiling the test programs. The computers have abundant memory (16GB and 128GB) so there are unlikely page swap outs in the experiment.

We have two implementations for testing. Firstly, we have TinySTM 0.9.5 [7] obtained from the Internet. TinySTM has a single common clock with clock-based validation and timestamp extension mechanism. When a transaction encounters a timestamp clock that is too new, it attempts to revalidate the read set before resorting to abort. Secondly, we have our TrC-MC implemented with flexible parameters. By default we

TABLE V
BENCHMARK SOFTWARE PARAMETERS

Software	Parameters
disjoint	(each thread increment a counter for 10^8 times.)
ssca2	-s20 -i1.0 -u1.0 -l3 -p3
genome	-g16384 -s64 -n16777216
vacation	-n4 -q60 -u90 -r1048576 -t4194304

have timestamp extension enabled and the number of zones equal to number of processors. With *sched_setaffinity* system calls, we lock the threads to their particular processors and zones. Threads in the same processor share information in the respective zone descriptor. The same implementation is capable mimicking other STM designs. By switching off the timestamp extension and setting number of zones to be number of processors, the system is very similar to the original TL2C. Therefore we will call this mimic version as TL2C in the rest of the evaluation.

B. TM Benchmark Programs

We put four applications into testing. *disjoint* is our self-brew benchmark, with transactions touching totally disjoint data. It reveals how many writing transactions a TM platform can handle. By its disjoint nature, there are not any aborts or timestamp extensions. *ssca2*, *genome* and *vacation* are the STAMP benchmarks [29] specially tailored for TinySTM. We pick the "real" problem size so each run takes several seconds to complete, ensuring timing accuracy and fairness. Through our experience [30], we know *ssca2* is difficult to scale because the transactions are short and frequent. *genome* is a gene-matching benchmark with moderate inter-thread data sharing. *vacation* is a company database simulation with random data access and is similar to TPC-C benchmark. The parameters given to the benchmark programs are shown on Table V. Each of the software is run for 4 times for data taking. Through the process, no anomalies (e.g., process crashing, abnormal data items) are detected.

C. Performance

Figure 2 shows the performance of some TM applications on two computers. In this paper we observe the performance in two metrics, the commit rate and abort rate. Commit rate indicates the relative speed of a platform to complete the same task. Abort rate indicates the amount of wasted work of a platform. Timestamp extension counters are available for TinySTM and TrC-MC. For comparison, we also have taken relative performance of the sequential counterparts that also come with the STAMP package.

Although *disjoint* involves simple, short and disjoint transactions only, the performance difference among different STMs is huge. TinySTM is the worst among the three test platforms. It cannot scale at all. It is because all the threads contend to access a single clock variable. TrC-MC is generally better in quad-processor platform while TL2C is better in dual-processor platform. This is probably because the two platforms perform differently in cache coherence. Nevertheless, supporting several 10^7 transactions per second on a system is useful enough in most situations, as real life applications would be at least slightly longer and less frequent.

The inability of TinySTM (and other single-clock STMs) is revealed completely on the *ssca2* benchmark. On the dual-processor system, TinySTM barely has 2 times absolute speedup when there are 16 threads running. On the quad-processor system, it simply cannot scale any faster than the sequential program, no matter how many threads are put into computation. Meanwhile, TrC-MC and TL2C performs relatively well, providing several times of absolute speedup. Note TrC-MC also has much less aborts when there are 32 or 64 threads running.

In *genome* benchmark, we see the distributed-clock STM are in slight disadvantage on the quad-processor platform. While we are unable to give a proved explanation, we notice that TrC-MC, being less distributed than TL2C, performs slightly better. We suspect the distributed clock allowed the threads to run faster, and create more transactional conflicts. As other researchers [31], [32], [33] and we have already noticed, sometimes stalling some threads leads to better performance. Note *genome* has the lowest commit ratio [30] among the four benchmarks we have on this paper.

In *vacation* benchmark, as the transactions are much less frequent, we observe the three STM platforms perform similarly. Meanwhile, similar to what Avni has noticed [12], TL2C has increasing abort rate and it is especially obvious in this benchmark. [3] TrC-MC has solved the issue by having the clocks less distributed, and also providing a timestamp extension mechanism. TinySTM and TrC-MC have similar protocols except the former has single-clock and later is distributed. While TinySTM virtually has no timestamp extension activated, TrC-MC has certain amount of extensions. This suggests the extensions are related to the distributed clock design. TL2C and TrC-MC are also similar except former does not have timestamp extension and has much more aborts. This reveals the extensions actually help reducing the abort rate.

VII. CONCLUSIONS

In this paper, we have discussed two design changes to a distributed clock-based software transactional memory system. We proved the correctness and evaluated the performance with several benchmarks. Compared to traditional single-clock systems, our solution performs much faster when there are frequent transactional commits. Compared to the previous distributed clock-based TL2C, our solution no longer suffers from the issue of increasing abort rate.

As a future work, we plan to investigate separating the timestamp table into several cache zones, evaluating whether it is a good mean to achieve a word-based distributed transactional memory system.

[3] Indeed, we remove the abort data points for 64 threads as they would distort the graph.

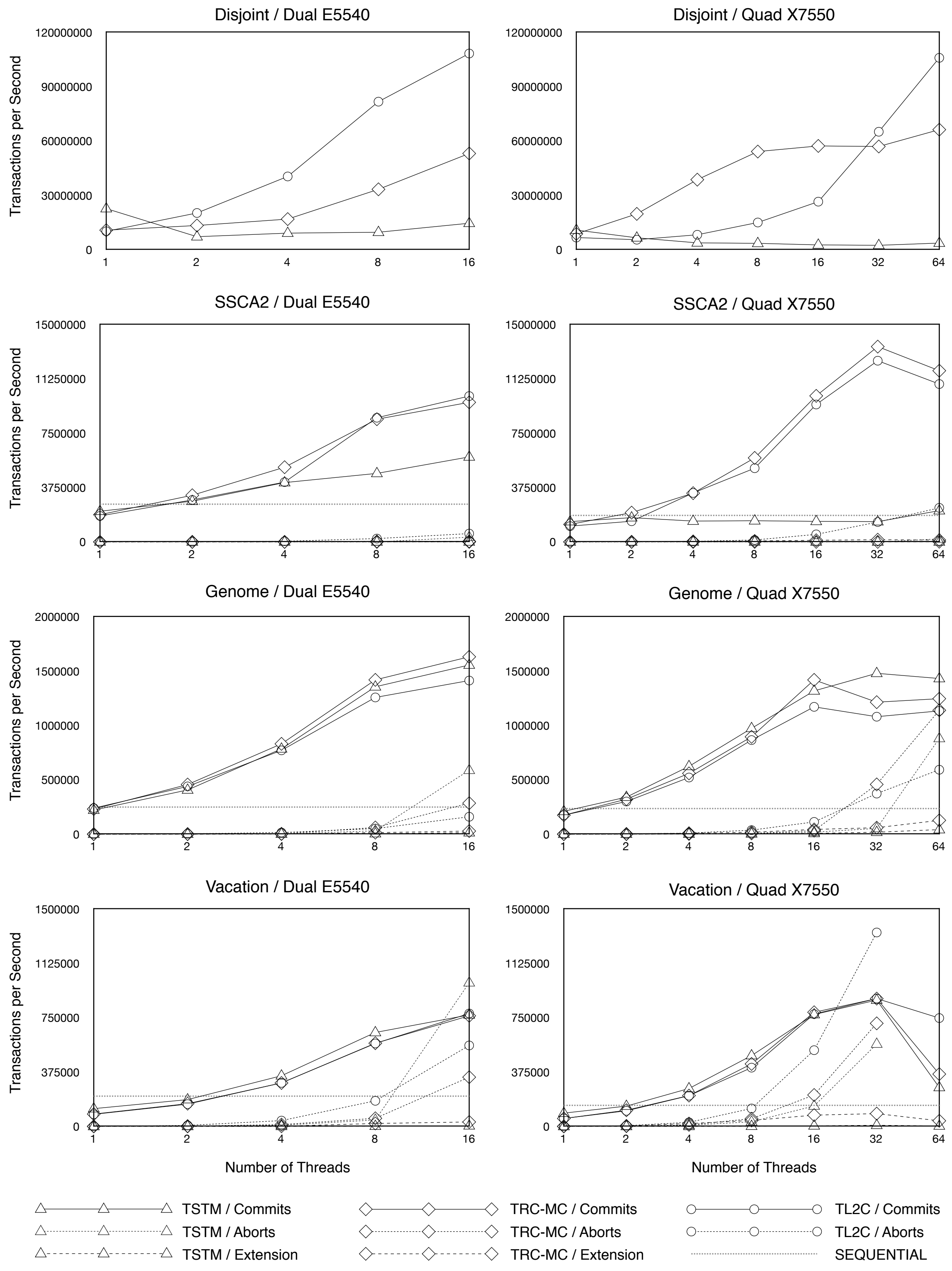

Fig. 2. Transaction commits and aborts of some TM applications on two computers

Acknowledgement

This research is supported by Hong Kong RGC Grant HKU7179/09E and Hong Kong UGC Special Equipment Grant SEG HKU09. We would also like to thank System Software Laboratory in National Tsing Hua University for lending one of the computers for this experiment.

References

[1] N. Shavit and D. Touitou, "Software transactional memory," in *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, 1995, pp. 204–213.

[2] C. Rossbach, O. Hofmann, and E. Witchel, "Is transactional memory actually easier," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010, pp. 47–56.

[3] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer III, "Software transactional memory for dynamic-sized data structures," in *Proceedings of the 21st annual Symposium on Principles of Distributed Computing*, 2003, pp. 92–101.

[4] V. Marathe, M. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. Scherer III, and M. Scott, "Lowering the overhead of nonblocking software transactional memory," in *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[5] D. Dice, O. Shalev, and N. Shavit, "Transactional locking II," in *Proceedings of the 20th International Symposium on Distributed Computing*, 2006, pp. 194–208.

[6] C. Wang, W. Chen, Y. Wu, B. Saha, and A. Adl-Tabatabai, "Code generation and optimization for transactional memory constructs in an unmanaged language," in *Proceedings of 2007 International Symposium on Code Generation and Optimization*, 2007, pp. 34–48.

[7] 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*, 2008, pp. 237–246.

[8] A. Dragojević, R. Guerraoui, and M. Kapalka, "Stretching transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Programming Language Design and Implementation*, 2009, pp. 155–165.

[9] A. Dragojevic, R. Guerraoui, and M. Kapalka, "Dividing transactional memories by zero," in *3rd ACM SIGPLAN Workshop on Transactional Computing*, 2008.

[10] T. Riegel, P. Felber, and C. Fetzer, "A lazy snapshot algorithm with eager validation," in *Proceedings of the 20th International Symposium on Distributed Computing*, 2006, pp. 284–298.

[11] T. Riegel, C. Fetzer, and P. Felber, "Time-based transactional memory with scalable time bases," in *Proceedings of the 19th annual ACM Symposium on Parallel Algorithms and Architectures*, 2007, pp. 221–228.

[12] H. Avni and N. Shavit, "Maintaining consistent transactional states without a global clock," in *Proceedings of the 15th international colloquium on Structural Information and Communication Complexity*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 131–140.

[13] H. Avni, "A transactional consistency clock defined and optimized," Ph.D. dissertation, Tel-Aviv University, 2009.

[14] T. Harris, J. Larus, and R. Rajwar, "Transactional memory," *Synthesis Lectures on Computer Architecture*, vol. 5, no. 1, pp. 1–263, 2010.

[15] D. Dice and N. Shavit, "TLRW: Return of the read-write lock," in *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, 2010, pp. 284–293.

[16] H. Attiya, E. Hillel, and A. Milani, "Inherent limitations on disjoint-access parallel implementations of transactional memory," in *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, 2009, pp. 69–78.

[17] M. F. Spear, A. Shriraman, L. Dalessandro, and M. L. Scott, "Transactional mutex locks," in *Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing*, 2009.

[18] 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*, 2010, pp. 67–78.

[19] M. Spear, M. Michael, and C. von Praun, "Ringstm: scalable transactions with a single atomic instruction," in *Proceedings of the 20th annual symposium on Parallelism in algorithms and architectures*, 2008, pp. 275–284.

[20] J. Gottschlich, M. Vachharajani, and J. Siek, "An efficient software transactional memory using commit-time invalidation," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, 2010, pp. 101–110.

[21] K. Manassiev, M. Mihailescu, and C. Amza, "Exploiting distributed version concurrency in a transactional memory cluster," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2006, pp. 198–208.

[22] P. Romano, N. Carvalho, and L. Rodrigues, "Towards distributed software transactional memory systems," in *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, 2008, pp. 4:1–4:4.

[23] C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C. Kirkham, and I. Watson, "Investigating software transactional memory on clusters," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*, 2008, pp. 1–6.

[24] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues, "D2stm: Dependable distributed software transactional memory," in *15th IEEE Pacific Rim International Symposium on Dependable Computing*, 2009, pp. 307–313.

[25] C. Kotselidis, M. Luján, M. Ansari, K. Malakasis, B. Kahn, C. Kirkham, and I. Watson, "Clustering jvms with software transactional memory support," in *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing*, 2010, pp. 1–12.

[26] T. Riegel, C. Fetzer, H. Sturzrehm, and P. Felber, "From causal to z-linearizable transactional memory," in *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, 2007, pp. 340–341.

[27] T. Riegel, C. Fetzer, and P. Felber, "Snapshot isolation for software transactional memory," in *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[28] R. Bocchino, V. Adve, and B. Chamberlain, "Software transactional memory for large scale clusters," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008, pp. 247–258.

[29] C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Standford transactional applciations for multi-processing," in *Proceedings of the IEEE International Symposium on Workload Characterization*, 2008, pp. 35–46.

[30] K. Chan, K. T. Lam, and C. L. Wang, "Adaptive thread scheduling techniques for improving scalability of software transactional memory," in *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing Networks*, 2011.

[31] M. Ansari, C. Kotselidis, K. Jarvis, M. Lujan, C. Kirkham, and I. Watson, "Adaptive concurrency control for transactional memory," in *Proceedings of the 1st workshop on Programmability Issues for Multi-Core Computers*, 2008.

[32] R. Yoo and H. Lee, "Adaptive transaction scheduling for transactional memory systems," in *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, 2009, pp. 169–178.

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