Enhancing Concurrency in Distributed Transactional Memory through Commutativity

Junwhan Kim, Roberto Palmieri, and Binoy Ravindran

ECE Department, Virginia Tech, Blacksburg, VA, 24061
{junwhan, robertop, binoy}@vt.edu

Abstract. Distributed software transactional memory is an emerging, alternative concurrency control model for distributed systems promising to alleviate the difficulties of lock-based distributed synchronization. We consider the multi-versioning (MV) model to avoid unnecessary aborts. MV schemes inherently guarantee commits of read-only transactions, but limit the concurrency of write transactions. In this paper we propose CRF (Commutative Requests First), a new scheduler tailored for enhancing concurrency of write transactions. CRF relies on the notion of commutative transactions, namely conflicting transactions that leave the state of the shared data-set consistent even if validated and committed concurrently. CRF is responsible to detect conflicts among commutative and non-commutative write transactions and then schedules them according to the execution state. We assess the goodness of the approach by an extensive evaluation of a fully implementation of CRF. The tests reveal that CRF improves throughput over a state-of-the-art DTM solution.

1 Introduction

Lock-based concurrency control suffers from programmability, scalability, and composability challenges [16]. Transactional memory (TM) promises to alleviate these difficulties sparing the programmers from the pitfalls of conventional manual lock-based synchronization, drastically simplifying the development of parallel and concurrent applications. With TM code, composed by read/write on shared objects, is organized as *memory transactions*, which optimistically execute, while logging changes made to accessed objects. Two transactions conflict if they access to the same object and at least one access is in write. When that happens, a contention manager is responsible to resolve the conflict by aborting one of them, yielding (the illusion of) atomicity. Aborted transactions are typically re-started, after rolling-back the changes. The contention manager can be supported by the *transactional scheduler*, that is the component responsible to determine an ordering among concurrent transactions so that conflicts are either avoided or minimized, thereby reducing abort rate and improving performance. Originally proposed to simplify concurrent programming in centralized environments, TM systems are being growingly employed in distributed settings (Distributed TM or DTM), motivated as an alternative to the more challenging distributed, lock-based, concurrency control. DTM can be classified based on

the system architecture: cache-coherent DTM (cc DTM) [14], in which a set of nodes communicate with each other by message-passing links over a communication network, and a cluster model (cluster DTM) [6, 17], in which a group of linked computers works closely together to form a single computer. Thanks to the simple distributed programming model provided by DTM, the programmers can focus on the implementation of the application logic, as it was centralized, putting the charge of distributed synchronization into the hands of DTM[18, 19]. With a single copy for each object, i.e., *single-version* STM (SV-STM), when a read/write conflict occurs between two transactions, the contention manager resolves the conflict by aborting one and allowing the other to commit, thereby maintaining the consistency of the (single) object version. SV-STM is simple, but suffers from large number of aborts [21]. In contrast, with multiple versions for each object, i.e., *multi-versioning* STM (MV-STM), unnecessary aborts of transactions, that could have been committed without violating consistency, are avoided [20]. Unless a conflict between operations to access a shared object occurs, MV-STM allows the corresponding transactions to read the object's old versions, enhancing concurrency. MV-STM has been extensively studied for multiprocessors [21, 12] and also for distributed systems [31]. MV-STM uses *snapshot isolation* (SI), which is weaker than serializability [24]. A transaction executing under SI operates on a snapshot taken at the start of the transaction. The transaction successfully commits if the objects updated by the transaction have not been changed externally since the snapshot was taken, guaranteeing that all read transactions will see a consistent snapshot. Many works [24, 25, 19] used SI for improving performance in centralized and distributed TM environments. Even though SI allows more concurrency among transactions respect to serializability, a write-write transaction's conflict under SI causes the transaction to abort. In write-intensive workloads, this conflict cannot be avoided because the concurrency of write transactions may violate SI.
In this paper, we address the problem of permitting multiple conflicting transactions to commit concurrently, in order to enhance concurrency of write transactions without violating SI in multi-version cc DTM for high performance. We propose a transactional scheduler that enables concurrency of write transactions, called Commutative Requests First (CRF). In order to do that, CRF exploits the notion of commutative operations. Two operations are named *commutative* if applying them sequentially in either order, they leave the objects accessed in the same state and both return the same values. A very intuitive example of commutativity is when two operations, $call1(X)$ and $call2(X)$, accessing both to the same object X but different fields of X (See Section 3 for discussion about commutativity). Thus, CRF checks whether write operations are commutative and lets them to validate and commit simultaneously. Unlike past STM works, that exploit high concurrency based on the commutativity property [13], CRF maintains a scheduling queue to identify commutative and non-commutative transactions, and could decide to allow all commutative transactions to commit first than the others, maximizing their concurrency. However, despite the significant performance gain obtainable adopting the idea of commu-

tativity transactions of CRF, there could be applications that do not admit such kind of commutativity. CRF addresses this issue permitting to the developer to explicitly specify non-commutative operations.

We implemented a full-working prototype of CRF in the Scala DTM framework, called HyFlow [29], and conducted extensive experimental studies using micro benchmarks (LinkedList and SkipList [22], as well as a real application benchmark (TPC-C [7]) typically used for assessing the performance of DTM. Our studies reveal that transactional throughput is improved by up to $5\times$ over a state-of-the-art DTM solution(DecentSTM [3]).

The rest of the paper is organized as follows. We outline the preliminaries and the system model in Section 2. We describe the CRF scheduler in Section 3. Implementation and experimental studies are reported in Section 4. We discuss the related work in Section 5 and Section 6 concludes the paper.

2 Preliminaries and System Model

We consider a distributed system which consists of a set of nodes $N = \{n_1, n_2, \cdots\}$ that communicate with each other by message-passing links over a network. Similar to [14], we assume that the nodes are scattered in a metric space.

Distributed Transactions. A set of *distributed transactions* $T = \{T_1, T_2, \cdots\}$ is assumed that share objects $O = \{o_1, o_2, \ldots\}$, which are distributed in the network. An execution of a transaction is a sequence of timed operations, reads and writes. An execution ends by either a commit (success) or an abort (failure). Each transaction has an unique identifier, and it is invoked by a node.

We consider data flow DTM model [14] where transactions are immobile and objects move to the node invoking transactions. Each node has a *TM proxy* that provides interfaces to the local application and to proxies at other nodes. When a transaction T_i at node n_i requests object o_j, the TM proxy of n_i first checks whether o_j is in its local cache. If the object is not present, the proxy invokes a distributed cache-coherence protocol (e.g., [8, 14]) to fetch o_j in the network. o_j may have multiple versions. The initial value of o_j is denoted by o_j^0. Let the version set of o_j be $\{o_j^0, o_j^1, \cdots\}$. Node n_k, holding the version set, checks whether the requested object version is in use by a local transaction T_k when it receives the request for o_j from n_i. If so, n_k's TM proxy invokes a contention manager to manage the conflict between T_i and T_k for the object version of o_j.

Atomicity, Consistency, and Isolation. We use the *Transactional Forwarding Algorithm* (TFA) [27] to provide *early validation* of remote objects, guarantee a consistent view of shared objects among distributed transactions, and ensure atomicity for object operations in the presence of asynchronous clocks. With early validation we refer to the fact that a transaction has already successfully validated its accessed objects before to commit. A validation in distributed systems includes global registration of object ownership. As an extension of the Transactional Locking 2 (TL2) algorithm [9], TFA replaces the central clock of TL2 with independent clocks for each node and provides a means to reliably establish the "happens-before" relationship between significant events. TFA is responsible for caching local copies of remote objects and changing its ownership. For completeness, we illustrate TFA with an example. In Figure 1, a transaction updates object o_1 at t_1 (i.e., local clock (LC) is 14) and four transactions (i.e., T_1, T_2, T_3, and T_4) request o_1 from the object holder. Assume that T_2 validates o_1 at t_2 and updates o_1 with LC=30 at t_3. Any read or write transaction (e.g., T_4), which has requested o_1 between t_2 and t_3, aborts. When write transactions T_1 and T_3 validate at times t_1 and t_2, respectively, T_1 and T_3 that have acquired o_1 with LC=14 before t_2 will abort, because LC is updated to 30.

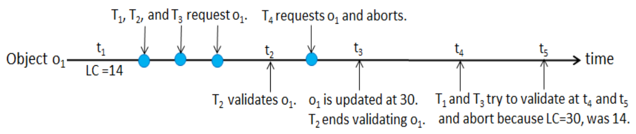

T1, T2, and T3 request o1.
T4 requests o1 and aborts.
Object o1
t1
LC =14
t2
t3
t4
t5
time
T2 validates o1.
o1 is updated at 30.
T2 ends validating o1.
T1 and T3 try to validate at t4 and t5
and abort because LC=30, was 14.

Fig. 1. An Example of TFA

3 Commutative Requests First in MV-TFA

Multi-Version TFA. In this section we present multi-version MV-TFA, our extension of TFA supporting SI. The basic idea is to record an event whenever requesting and acquiring an object. Let n_i denote a node invoking a transaction T_i. We define two types of events: (1) *Request*($\mathrm{Req(n_i, o_j)}$) representing the request of object o_j from node n_i; (2) *Acquisition*($\mathrm{Acq(n_i, o_j)}$) indicating when node n_i acquires object o_j. Figure 2 shows an example execution scenario of MV-TFA. We use the same style in the figure as that of [26]. The solid circles indicate write operations and the empty circles represent read operations. Transactions' evolution is represented on horizontal lines with the circles. The horizontal line corresponding to the status of each object describes the time domain. The dotted line indicates which node requests an object from where. Assume that transactions T_0 and T_1 invoked on nodes n_0 and n_1 commit after writing o_1^0 and o_2^0, respectively. Let transactions T_2, on node n_2, and T_3, on node n_3, request objects o_1 and o_2 from nodes n_0 and n_1, respectively. Node n_1 holds the list of versions of o_2. After that, T_3 requests o_1 from n_0 and subsequently T_4 requests o_2 from n_1. Thus, n_1 records the events $Acq(n_3, o_2^0)$ and $Acq(n_4, o_2^0)$. Then T_4 updates o_2 creating a new version o_2^1. When T_4 validates o_2^1 to commit, $Acq(n_4, o_2^0)$ is removed from the events log of n_3, and T_3 has forced to abort because in the n_3's log there is another request ($Acq(n_3, o_2^0)$) on the same object o_2. The presence of this entry in the log means that T_3 has not yet completed, so T_4 definitively commits before that T_3 validates o_2, invalidating the object o_2^0 accessed by T_3. As a consequence of T_4 commitment, node n_4, which invokes T_4, receives the versions o_2^0 and o_2^1 of object o_2. Now, after the commit of T_4, T_2 requests o_2 with the value $\mid t_4$ - $t_2 \mid$ from n_4. It replies with the version o_2^0 instead of the newly o_2^1 because o_2^0 has been updated at time t_1 to T_2, because $\mid t_4 - t_3 \mid < \mid t_4$ - $t_2 \mid < \mid t_4 - t_1 \mid$. Using this mechanism, T_2 can access to a consistent snapshot that is not affected by a write operation by T_4, instead of

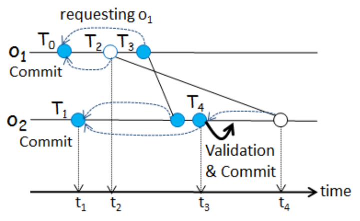
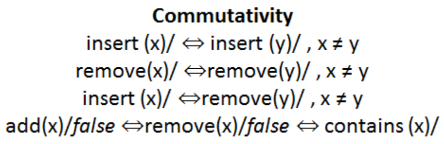


Fig. 2. Example of MV-TFA

be aborted due to T_4 's write. This is how MV-TFA ensures SI.

CRF Scheduler Design. MV-TFA shows how to enhance performance in case of workload characterized by mostly read transactions, exploiting multi-versions. In this subsection we focus on how to schedule write transactions concurrently minimizing the abort rate and increasing the parallelism.



Commutativity

insert (x)/ ⇔ insert (y)/ , x ≠ y

remove(x)/ ⇔remove(y)/ , x ≠ y

insert (x)/ ⇔remove(y)/ , x ≠ y

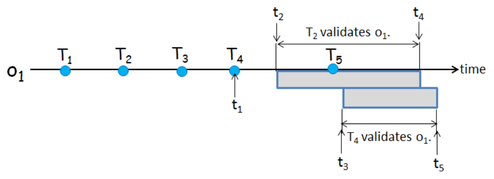
add(x)/*false* ⇔remove(x)/*false* ⇔ contains (x)/

Fig. 3. Specification of a Set

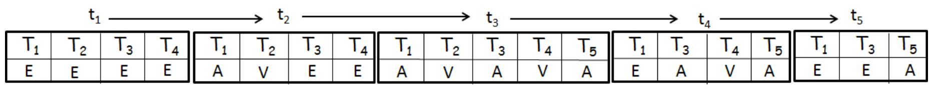
When a transaction T_1 at node n_1 needs object o_1 for an operation, it sends a request to the o_1's object owner. If the operation is read, a version of o_1 is sent to n_1. If the operation is write, we consider two possible cases in terms of o_1. *(A)* The first case happens when other transactions may have requested o_1 but no transaction has validated o_1. In this case, a version of o_1 is sent to n_1 and T_1's request moves into the scheduling queue of the o_1's owner. *(B)* The second case is when another transaction T_2 is validating o_1. In this case, unless T_2 and T_1 commute, T_1 will abort and T_1's request also moves to the scheduling queue. If T_2 and T_1 commutes, o_1 is sent to n_1 and T_1's request moves to the scheduling queue. The o_1's owner maintains the scheduling queue to execute commutative transactions concurrently. Accordingly, the non-commutative transactions will be executed serially. To better assess CRF, we use it to implement the specification of a *Set* provided by [13]. We recall that a *Set* is a collection of items without duplications in which the following operations are provided: $add(x)$, $remove(x)$ and $contains(x)$ where x is the item of the *Set* accessed. Figure 3 summarizes *Set* operations' commutativity according to [13]'s definition. In the specification illustrated in Figure 3, operations $insert(x)$, $insert(y)$, and $insert(z)$ commutes if $x \neq y \neq z$. Multiple write transactions may be invoked concurrently on the *Set*. CRF identifies commutative and non-commutative transactions and gives to the commutative transactions a chance to validate concurrently an object first. However, if we consider the specification of the *Set*, in which the are no commutative operations declared, and we encapsulate the *Set* into an object (o_1) and we consider the above operations as transactions, then typical concurrency control does not permit to validate and commit concurrently more than one transaction performing an update on the object (namely updating the *Set*). Conversely, by Figure 3 is clear that multiple update transactions can be validated concurrently whether they access to different items in the set. The scheduling queue holds requests for those operations. If multiple transactions have requested the same version of o_1, CRF allows the commutative transactions to concurrently validate o_1. Meanwhile, many commutative transactions may validate o_1. This could bring non-commutative transactions to "starve" on o_1. Thus, CRF alternates between periods (called *epochs*), in which it privileges the validation of a group of commutative transactions, with others in which it prefer to validate the non-commutative ones. In this way, CRF handles conflicts between commutative and non-commutative transactions. Although epochs contain commutative transactions, these transactions do not commute with the transactions of the next epoch in the chronological sequence. The terminology

"commutative" and "non-commutative" epoch distinguishes between these two epochs. Thus, in commutative epoch, commutative transactions validate o_1 and then in the next (i.e., non-commutative) epoch, non-commutative transactions, excluded in the previous commutative epoch, can validate o_1. If a transaction starts validating o_1, its commutative transactions are also allowed to validate o_1 but its non-commutative transactions abort. The non-commutative transactions will resume after the commutative transactions commit.
CRF checks for whether different operations commute at the level of semantics. Even when commutative operations concurrently update the object, the object preserves a consistent state, ensuring SI. There are two purposes for processing commutative requests first. First, MV-TFA ensures concurrency of read transactions, and CRF is responsible to detect conflicts among commutative and non-commutative write transactions, reducing the number of conflicts. This leads to higher concurrency. Second, CRF alleviates contention when many write transactions are invoked. Even though a conflict between two write transactions occurs, all subsequent commutative transactions are scheduled first. Non-commutative transactions restart simultaneously after the commutative transactions complete, so CRF avoids further conflicts, decreasing contention.

Illustrative Example. Figure 4 shows a scenario of CRF. The write trans-



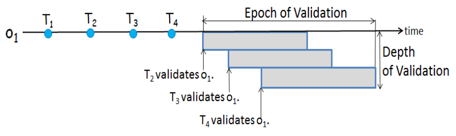
(a) Requests of Five Transactions and Validation of Two Transactions for Object o_1.


t_1
t_2
t_3
t_4
t_5
T_1 T_2 T_3 T_4 / E E E E
T_1 T_2 T_3 T_4 / A V E E
T_1 T_2 T_3 T_4 T_5 / A V A V A
T_1 T_3 T_4 T_5 / E A V A
T_1 T_3 T_5 / E E A

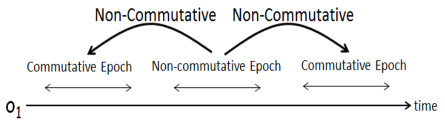
(b) Scheduling Queue Located in o_1 Object Owner. The scheduling queue consists of two rows: Enqueued Transactions and State of the Transactions. V (Validation), A (Abort), and E (Execution)

Fig. 4. A Scenario of CRF

actions T_1=$insert(x)$, T_2=$remove(x)$, T_3=$insert(y)$ and T_4=$remove(y)$ request concurrently o_1 from its owner. The transactions obtain the version of o_1. The state of the scheduling queue at t_1, illustrated in Figure 4(b), shows that the transactions are all executing. At t_2, T_2 starts validating o_1. Consequently, T_1 aborts because T_1 and T_2 do not commute. Conversely, T_3 and T_4 can still execute because they are commutative with T_2. Then T_5=$remove(x)$ requests o_1 during the validation of T_2 and immediately aborts because T_5 and T_2 do not commute. At t_3, T_4 starts validating o_1 and T_3 aborts because T_3 and T_4 do not commute. Thus, T_2 and T_4 concurrently validate o_1. When T_2 ends validation (i.e., commits) at t_4, the version updated by T_2 is sent to the non-commutative transaction T_1, and T_1 starts executing. Even though T_5 is a non-commutative transaction of T_2, only T_1 starts to avoid a conflict between non-commutative transactions. Finally, the version updated by T_4 at t_5 is sent to T_3. T_1 and T_3 may validate o_1 concurrently because they commute. Figure 5(a) shows that the val-



(a) Epoch and Depth of Validation.



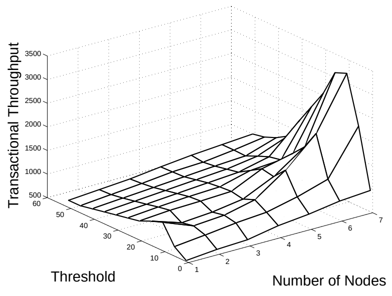
(b) Epochs of Validation

Fig. 5. Epoch-based CRF

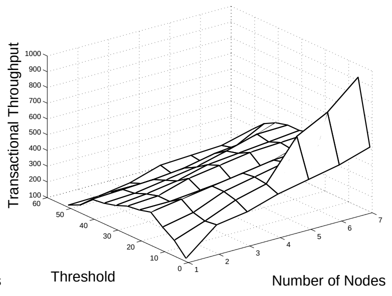
idation of commutative transactions may not be completely overlapping, so the period of validation may be stretched. This may lead to the deferred execution of non-commutative transactions. To prevent this, we define a new parameter, called *depth of validation*, namely the number of transactions involved in the validation. Figure 5(a) indicates 3 for that depth, meaning that the commits of three transactions mark the end of the epoch. Non-commutative transactions will start after the epoch. Figure 5(b) illustrates the relationship of epochs. In each epoch, commutative transactions concurrently participate in validation. At the end of the epoch, their non-commutative transactions held in a scheduling queue, restart. Non-commutative transactions will validate in the next epoch.

4 Implementation and Experimental Evaluation

Implementation. We implemented CRF on MV-TFA using Scala's actor model for Java Virtual Machine. The actor model prohibits sharing memory by en-



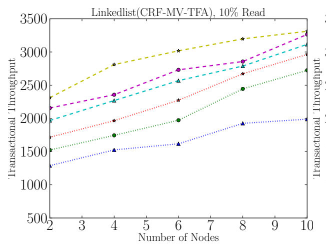
(a) LinkedList



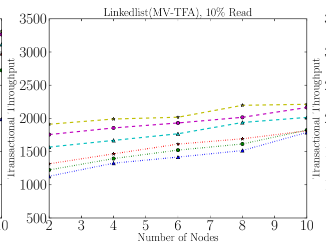
(b) TPC-C

Fig. 6. Throughput Varying Thresholds

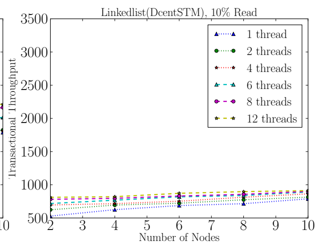
capsulating mutable state inside light-weight sequential constructs called actors and it become popular with the advent of the Erlang programming language. Since then, many languages (e.g., Google Go) have embraced this model.
Commutativity of Benchmarks. We assess the performance of CRF using LinkedList and SkipList as micro-benchmarks and a TPC-C [7] as real-application benchmark. Regarding the commutativity in micro-benchmarks, the *Set* (as introduced in Section 3) can be implemented with LinkedList and SkipList [13], so we rely on the definition of commutativity in Figure 3. Regarding TPC-C,



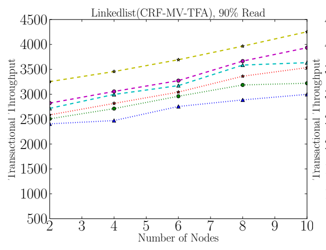
(a) CRF-MV-TFA, 10% Read



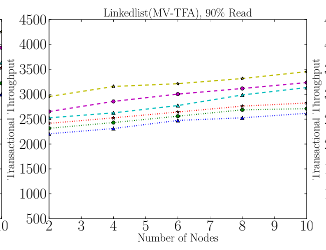
(b) MV-TFA, 10% Read



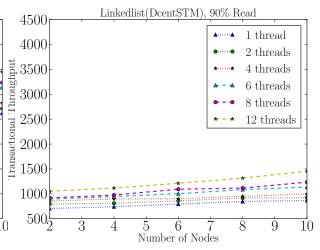
(c) DecentSTM, 10% Read



(d) CRF-MV-TFA, 90% Read



(e) MV-TFA, 90% Read



(f) DecentSTM, 90% Read

Fig. 7. Throughput of CRF, MV-TFA, and DecentSTM Using LinkedList.

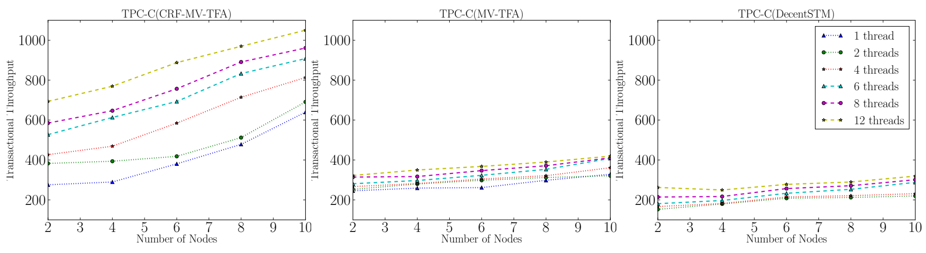
the write transactions consist of update, insert, and/or delete operations accessing a database of nine tables maintained in memory. Each row in the tables has a unique key. Multiple operations commute if they access to a row (or object) with the same key and modify different columns. We rely on explicit annotations provided by the programmer, indicating the fields accessed by each transaction profile. We configured the benchmark with a limited number of warehouses (#4) in order to generate high conflicts. We recall that, in data flow model, objects are not bound on fixed nodes but move, increasing likelihood of conflicts.

Experimental Setup. Our test-bed consists of 10 nodes connected via a switched 1 Gigabit network connection. Each node is comprised of 12 Intel Xeon 1.9GHz processor cores. We use the Ubuntu Linux 10.04 server OS. We measured the *transactional throughput* (number of committed transactions per second). To manage garbage collection, versions that are no longer accessible, need to be marked. Unlike multiprocessors, determining old versions for live transactions in distributed systems incurs communication overheads. Thus, we consider a threshold-based garbage collector [5], which checks the number of versions and disposes the oldest if the number of versions exceeds a pre-defined threshold. We consider threshold 4 for measuring the basic event model's throughput, because the observed that the speed-up is relatively less increased after threshold.

Finding a Depth. The large number of concurrent validations may lead to a significant scheduling overhead due to delayed non-commutative transactions. For the balance of commutative and non- requesting transactions, we consider a threshold-based control, switching the next epoch when either a depth or a number of non-commutative transactions enqueued meets a predefined threshold, called $MaxD$. Figure 6 shows throughput moving the $MaxD$ from 1 to 50.

By the plot is clear that CRF's throughput is not improved after $MaxD$=10 for LinkedList and $MaxD$=5 for TPC-C due to the increasing number of non-commutative transactions aborted. With the previous values of $MaxD$, CRF reaches its maximum throughput, so we used those for the experiments.

Evaluation. Figures 7,9 show the throughput of CRF, MV-TFA and De-

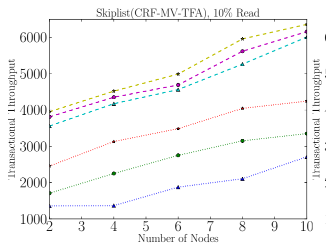


(a) CRF-MV-TFA (b) MV-TFA (c) DecentSTM

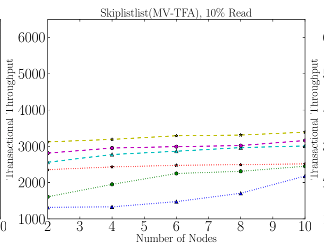
Fig. 8. Throughput of CRF, MV-TFA, and DecentSTM Using TPC-C

centSTM using LinkedList(Figure 7) and SkipList(Figure 9) benchmarks. The legend has to be considered for all the plots and shows the colors differentiate for number of running threads. Each micro-benchmark has been evaluated using two workloads representative of read intensive (10% writes and 90% reads) and write intensive (90% writes and 10% reads) scenarios. The tests have been performed varying the number of nodes and the number of threads per node. Each thread submits requests to the distributed system. Summarizing, we span scenarios from 2 up to 120 concurrent threads in the system. This allows to exhaustive assess the behavior of CRF. The comparison between CRF and MV-TFA shows how much CRF enhances the concurrency of write transactions. For the LinkedList and SkipList, the new value to add or delete is randomly selected using a uniform distribution. According to the increasing number of threads and nodes, CRF performs better due to the detection of a large number of commutative operations. Even though the throughput of CRF is slightly better than MV-TFA in scenario characterized by most read-only transactions (due to the limited number of commutative write operations), the maximum gain of CRF against competitors is reached in write-intensive workload where CRF exploits the ability to validate and commit concurrently conflicting transactions. In additional the plot revels that, in write dominated workload, CRF scales better than MV-TFA and DecentSTM. In fact, in contrast with CRF, their performance stall increasing the number of concurrent threads in the system. This is also confirmed by the plot in Figure 9(a) and 9(b) where CRF outperforms MV-TFA by as much as 2×. As a competitor, DecentSTM [3] is based on a snapshot isolation algorithm, which requires searching the history of objects to find a valid snapshot. This algorithm incurs a significant overhead. Thus, we observe that the transactional throughput of DecentSTM is not improved as long as requesting nodes increase. Our evaluations reveal that CRF improves throughput over MV-TFA and DecentSTM by as much as (average) 2× and 3× under 10% read transactions, respectively. Further, our evaluations show that

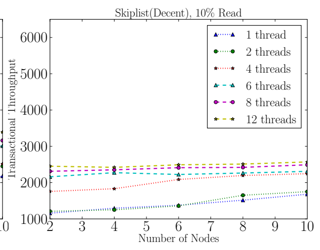
MV-TFA outperforms DecentSTM in throughput as much as 2×. Figure 8 shows the throughput of CRF, MV-TFA, and DecentSTM using TPC-C benchmark. We used the amount of read and write transactions that the specification of TPC-C recommends. TPC-C benchmark accesses large tables to read and write values. Due to the non-negligible transaction execution time, scheduling commutative operations highly impacts the overall performance. In fact, the conflicting transactions generated by the benchmark are well managed by CRF and this results observing that CRF performs better than DecentSTM as much as 5× over 10 nodes.



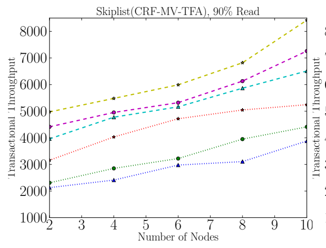
(a) CRF-MV-TFA, 10% Read



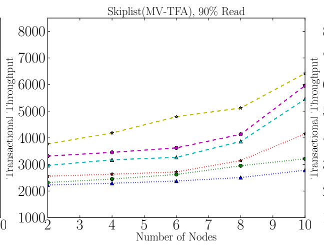
(b) MV-TFA, 10% Read



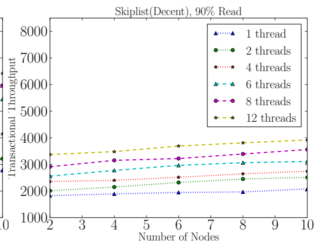
(c) DecentSTM, 10% Read



(d) CRF-MV-TFA, 90% Read



(e) MV-TFA, 90% Read



(f) DecentSTM, 90% Read

Fig. 9. Throughput of CRF, MV-TFA, and DecentSTM Using SkipList.

5 Related Work

Transactional scheduling has been explored in a number of multiprocessor STM efforts [11, 1, 30, 10, 2]. In [11], is described an approach that dynamically schedules transactions based on their predicted read/write access sets. In [1], the authors discuss the Steal-On-Abort transaction scheduler, which queues an aborted transaction behind the non-aborted transaction, and thereby prevents the two transactions from conflicting again. In [30] is presented Adaptive Transaction Scheduler (ATS), that adaptively controls the number of concurrent transactions based on the contention intensity: when the intensity is below a threshold, the transaction begins normally; otherwise, the transaction stalls and does not begin until dispatched by the scheduler. CAR-STM scheduling approach [10] uses per-core transaction queues and serializes conflicting transactions by aborting one and queueing it on the other's queue, preventing future conflicts. CAR-STM pre-assigns transactions with high collision probability (application-described) to the same core, and thereby minimizes conflicts. In [4] has been proposed the Proactive Transactional Scheduler (PTS). This scheme detects hot spots of contention that can degrade performance, and proactively schedules affected transactions around the hot spots. In [2] has been presented the BIMODAL scheduler, which targets read-dominated and bimodal (i.e., those with only early-write and read-only) workloads. BIMODAL alternates between "writing epochs" and "reading epochs" during which writing and reading transactions are given priority, respectively, ensuring greater concurrency for reading transactions. Steal-On-Abort, CAR-STM, and BIMODAL enqueue aborted transactions to minimize future conflicts in SV-STM. In contrast, CRF only enqueues non-commutative transactions that conflict with commutative transactions. The purpose of enqueuing is to prevent contending transactions from requesting all objects again. Thus, CRF also minimizes conflicts, but the overhead of CRF's scheduling is lower than the others because the number of enqueued transactions is smaller. ATS and PTS determine contention intensity and use it for contention management. Unlike these schedulers which are designed for multiprocessors, CRF maintains contention monitoring only between commutative and non-commutative write transactions, alleviating some of the overhead of contention management. In terms of commutativity, in [15] has been used a similar approach of CRF in order to running in parallel independent parts of the code.

It is important to note that, MV-STM has been extensively studied for multiprocessors and for distributed systems. MV increases concurrency by allowing transactions to read older versions of shared data, thereby minimizing conflicts and aborts. For example, in [23] is presented a dependency-aware transactional memory (DATM) for multiprocessors, where transaction execution is interleaved, and show substantially more concurrency than two-phase locking. Replicated object models for DTM have been studied in [17, 28], but these efforts also do not consider scheduling.

6 Conclusions

We presented a commutativity-based transactional scheduler for multi-version DTM, called CRF. CRF focuses on how to enhance concurrency of write transactions in multiversioning schemes ensuring SI, where write transactions are exposed to a high probability of conflicts. Our key idea is to detect a conflict between commutative and non-commutative write transactions and allow the first ones to commit concurrently before the others. CRF's design shows how commutativity-based scheduling impacts throughput in DTM. Our experimental evaluation shows that CRF enhances throughput over a state-of-the-art DTM solution, by 3 and 5× using micro-benchmarks and real-application.

Acknowledgements

This work is supported in part by US National Science Foundation under grants CNS 0915895, CNS 1116190, CNS 1130180, and CNS 1217385.

References

1. Mohammad Ansari, Mikel Lujn, et al. Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In *HiPEAC*, 2009.
2. Hagit Attiya and Alessia Milani. Transactional scheduling for read-dominated workloads. In *OPODIS*, pages 3–17, Berlin, Heidelberg, 2009. Springer-Verlag.
3. A. Bieniusa and T. Fuhrmann. Consistency in hindsight: A fully decentralized STM algorithm. In *IPDPS, 2010 IEEE*, pages 1 –12, april 2010.
4. G. Blake, R.G. Dreslinski, and T. Mudge. Proactive transaction scheduling for contention management. In *Microarchitecture, 2009.*, pages 156 –167, dec. 2009.
5. Stefan Büttcher and Charles L. A. Clarke. Indexing time vs. query time: trade-offs in dynamic information retrieval systems. CIKM '05, pages 317–318. ACM, 2005.
6. M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D2STM: Dependable distributed software transactional memory. In *PRDC*, nov 2009.
7. TPC Council. "tpc-c benchmark, revision 5.11". Feb 2010.
8. Demmer and Herlihy. The arrow distributed directory protocol. In *DISC 98.*
9. David Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC*, 2006.
10. Shlomi Dolev, Danny Hendler, and Adi Suissa. CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory. In *PODC*, 2008.
11. Aleksandar Dragojević, Rachid Guerraoui, et al. Preventing versus curing: avoiding conflicts in transactional memories. In *PODC '09*, pages 7–16, 2009.
12. Sérgio Miguel Fernandes and João Cachopo. Lock-free and scalable multi-version software transactional memory. PPoPP '11, pages 179–188. ACM, 2011.
13. Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. PPoPP '08, pages 207–216. ACM, 2008.
14. Maurice Herlihy and Ye Sun. Distributed transactional memory for metric-space networks. *Distributed Computing*, 20(3):195–208, 2007.
15. M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, 2007.
16. James R. Larus and Ravi Rajwar. *Transactional Memory.* M. and Claypool, 2006.
17. Roberto Palmieri, Francesco Quaglia, and Paolo Romano. Osare: Opportunistic speculation in actively replicated transactional systems. In *SRDS*, 2011.
18. S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *ICDCS*, 2012.
19. Sebastiano Peluso, Paolo Romano, and Francesco Quaglia. Score: A scalable one-copy serializable partial replication protocol. In *Middleware*, pages 456–475, 2012.
20. Keidar Perelman. On avoiding spare aborts in transactional memory. In *SPAA 09.*
21. Keidar Perelman, Fan. On maintaining multiple versions in STM. In *PODC 10.*
22. W Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun.*, 1990.
23. Hany E. Ramadan et al. Dependence-aware transactional memory for increased concurrency. In *MICRO*, pages 246–257, 2008.
24. T. Riegel, C. Fetzer, and P. Felber. Snapshot isolation for software transactional memory. In *ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing '06.*
25. T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *SPAA*, 2007.
26. T. Riegel, C. Fetzer, H. Sturzrehm, and P. Felber. From causal to z-linearizable transactional memory. In *PODC*, 2007.

27. M. Saad and Binoy R. Supporting STM in distributed systems: Mechanisms and a Java framework. In *ACM SIGPLAN Workshop on Transactional Computing '11*.
28. Nicolas Schiper, Pierre Sutra, and Fernando Pedone. P-store: Genuine partial replication in wide area networks. In *SRDS*, pages 214–224, 2010.
29. A. Turcu and B. Ravindran. Hyflow2: A high performance distributed transactional memory framework in scala. http://hyflow.org.
30. Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. In *SPAA*, pages 169–178, 2008.
31. Bo Zhang and Binoy Ravindran. Brief announcement: on enhancing concurrency in distributed transactional memory. PODC '10.