©2010 International Journal of Computer Applications (0975 - 8887)
Volume 1 – No. 22

Clustered Checkpointing and Partial Rollbacks for Reducing Conflict Costs in STMs

Monika Gupta
IBM India Research Lab
New Delhi, India

Rudrapatna K Shyamasundar
Tata Institute of Fundamental Research, India

Shivali Agarwal
IBM India Research Lab
New Delhi, India

ABSTRACT
A Software Transactional Memory is a concurrency control mechanism that executes multiple concurrent, optimistic, lock-free, atomic transactions, thus alleviating many problems associated with conventional mutual exclusion primitives such as monitors and locks. With the advent of massive multi-cores, more transactions can be initiated concurrently, however resulting in an increase in the percentage of conflicting transactions. Each time a transaction conflicts, it imposes a significant cost on the system, originating from the need to abort and redo all the operations, including the costly shared memory read operations, thus making the overall system significantly heavy and impractical. We present an algorithm, *Clustered Checkpointing and Partial Rollback (CCPR)*, for reducing the conflict costs of transactions in the face of increasing conflicts. The algorithm is based on intelligent checkpointing of transactions as they proceed, and, in case of conflict, rolling them back to a safe, consistent, intermediate checkpoint, thus reducing conflict costs. The intelligence of the algorithm lies in the fact that as conflicts decrease, the checkpointing costs go low, however, when conflicts increase, the checkpointing costs increase but are still pretty much less than the amount of savings obtained by the partial rollback of the conflicting transactions. We simulated several applications in the CCPR framework and found that it can result in as good as 17% reduction in the conflict costs originating from the need to redo all the shared memory read operations.

General Terms
Concurrent Programming, Software Transactional Memory

Keywords
Software Transactional Memory, Clustered Checkpointing, Dynamic Clustering, Probabilistic Clustering, Partial Rollback

1. INTRODUCTION
Recent advances in multi-core architectures demand efficient synchronization mechanisms to achieve performance scaling. The conventional primitives such as locks if coarse grained suffer from problems of scalability, while fine grained locks become difficult to program as it becomes difficult to visualize deadlocks due to interleaving executions. With multiprocessing becoming ubiquitous and concurrent applications a norm, various solutions for easy-to-program, scalable and efficient synchronization mechanisms are being sought. There has been a growing consensus that *transactions* can provide a simple, powerful mechanism for synchronization over multiple objects. Sequences of object references can be grouped to form transactions, and each such transaction can be treated as an atomic execution unit. Programmers can focus on the atomicity requirements rather than the implementation details of synchronization. Infact, some of the futuristic parallel languages like X10, being targeted for high performance and productivity have already incorporated the notion of *atomic* computation as a language construct. These explorations have lead to the abstraction of *Transactional Memory* (TM) [1] as a realization for such atomic units of computation.

A TM is a concurrency control mechanism that executes multiple concurrent, optimistic, lock-free, atomic transactions, thus alleviating many problems associated with explicit locking. Shared memory acts as a large database which is shared by multiple isolated transactions / execution threads. TM guarantees atomicity and isolation of the sequential code executed within a transaction by appropriately committing/aborting them. A TM thus allows programmers to focus on the atomicity requirements rather than the implementation details of the synchronization. TMs can be classified as either STMs (Software TM) or HTMs (Hardware TM), based on whether the transactional semantics are implemented in software or hardware. We consider in this paper a STM system, and propose an algorithm for its realization.

Several aspects have been used to classify existing STM algorithms, some of which are:

- **When does a transaction actually update the desired shared objects?** - Eager versioning STMs are typically lock-based blocking implementations, where transactions modify data in-place by using logs. Lazy versioning STMs are non-blocking implementations, where transactions usually execute by making a private working copy of the shared objects and when completed, swap their working copy with the global copy thus assuring that both committing and aborting are light-weight operations.
- **When does a transaction detect a conflict with another transaction in the system?** –While in Eager Conflict Detecting STMs, conflicts are detected as transactions proceed, in Lazy Conflict Detecting STMs, conflicts are detected at commit time.
- **How do transactions commit themselves? -** A commit operation in a STM is either a lock-free operation based on indirection and compare-and-swap (CAS), or a locking operation. A locking operation uses either Encounter-Time Locking or Commit-Time Locking. In encounter time locking, memory writes are done by first temporarily acquiring a lock

for a given location, writing the value directly, and then logging it in the undo log. Commit-time locking uses two-phase locking scheme, i.e., it locks all memory locations during the first phase (called acquire phase) and updates and unlocks them in the second phase (called commit phase).

Different implementations of STMs [2] make tradeoffs that impact both performance and programmability. Calin et. al. in [3], explored the performance of a two STM algorithms and observed that the overall performance of TM was significantly worse at low levels of parallelism. In this paper, we propose an algorithmic extension to one of the STM algorithms they explored, i.e., the *global version number* (gv) algorithm. We show through simulation results that the new algorithm we propose is efficient than the original one, especially when the percentage of conflicting transactions is moderate to high.

The rest of the paper is organized as follows. We discuss the CCPR algorithm in Section 2. In section 3, we present CCPR's simulation results. Section 4 discusses the related work, and we wrap up in section 5 with conclusions and future work.

2. THE CCPR ALGORITHM

2.1 The Baseline - global version number (gv) algorithm

Calin et. al. [3] studied the performance of two STM algorithms – one that fully validates (fv) the read set after each transactional read, and the other that uses a global version number (gv) to avoid the full validation. While the fv algorithm provides more concurrency at a higher price, the gv algorithm is deemed as one of the best tradeoffs for STM implementations. We assume the gv algorithm as our baseline algorithm.

Figure 1 is taken from [3]; it details out which operations in a STM cause maximum overheads. As is clear from the figure, the overheads of the transactional reads dominate other operations because of the relatively higher frequency of these operations. Although the gv algorithm does reduce the read overheads as compared to the fv algorithm, still the read operations contribute significantly to the overall overheads, and this worsens when the transactions conflict with each other, and upon abort start afresh from the beginning. Having discussed all this, we next discuss the CCPR algorithm and illustrate how the algorithm reduces the shared read overheads without incurring too much of overheads itself.

2.2 The CCPR algorithm

The CCPR algorithm extends the gv algorithm by appropriately checkpointing the transactions as they execute in their local workspace, and in case of a conflict, uses the checkpoint logs to identify a safe, consistent, intermediate checkpoint to partially rollback to. For checkpointing purposes, the algorithm abstracts the shared memory as a set of shared objects which in its finest form can be a simple data type such as an integer, float, character etc., or, it can be coarse as a user-defined data type, e.g. a link-list node. The CCPR algorithm along with its data structures and operations is presented below.

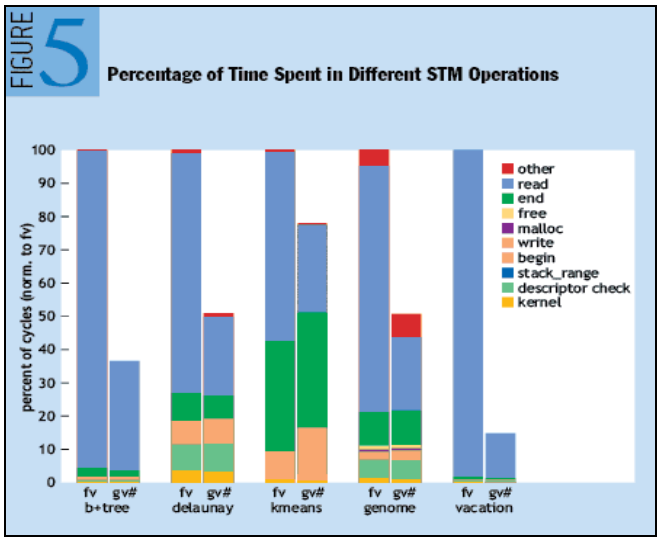

Figure 1. Percentage of time spent in different STM operations

2.2.1 Data Structures

Each shared object in the shared memory is augmented with a 4-bit *conflict probability* value which increases as and when the shared object is involved in a conflict, and is reset to zero when no transactions are reading it. Transactions create new checkpoints for only those shared objects which have a good probability of ending up in a conflict; otherwise the new checkpoint is clustered with the previous checkpoint, thus reducing overheads.

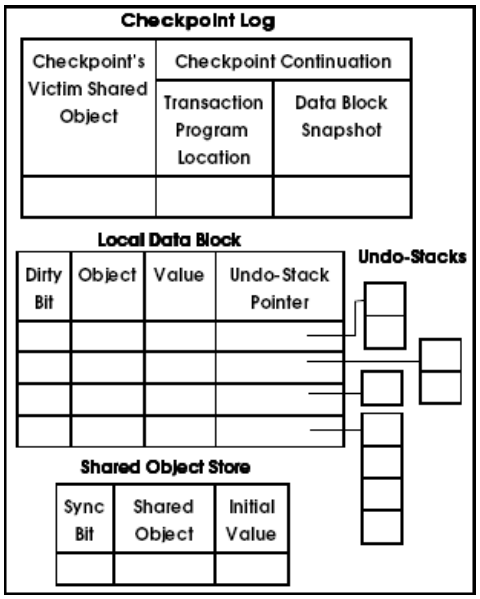

Figure 2. Transaction's workspace

A transaction's workspace is shown in figure 2. Each transaction maintains a *local data block*, a *shared object store*, and a

checkpoint log. Whereas a local data block stores the current values of all the local/shared objects being currently used by the transaction, the shared object store stores the initial values of only the shared objects as originally read from the shared memory.

Shared Object Store – Each entry in the shared object store contains the following: (1) the shared object, (2) its initial value as read from the shared memory, and, (3) a sync-bit indicating whether or not this value is in-sync with the object's current value in the shared memory.

Values of shared objects read from shared memory are updated in the shared object store (and also in the local data block) and the corresponding sync-bit is set to "1" to indicate an in-sync value. As a transaction conflicts, some of these shared objects become victims of conflict and their values go out-of-sync (a "0" sync-bit) with the corresponding values in the shared memory, at which point the transaction needs to re-start from a suitable checkpoint and re-read these objects from the shared memory.

Local Data Block - Each entry in the local data block contains the following: (1) the local/shared object, (2) its current value in the transaction, (3) a dirty-bit indicating whether or not the object's value has been updated since the last checkpoint, and, (4) a pointer to the object's undo-stack. Each local/shared object in the transaction maintains an undo-stack to trace the different values assigned to it as the transaction proceeds.

As a transaction proceeds, various read/write requests are served as follows:

- All shared object read requests are directed to the local data block, if not served there, are redirected to the shared object store, and if not served there also, are redirected to the shared memory and subsequently cloned in the shared object store and local data block for further read/write requests.
- All local object read requests get served through the local data block.
- All writes are done in the local data block and the corresponding dirty-bits for the objects being written are set.

Checkpoint Log - A checkpoint log is essentially a variant of an undo-log, and is used for partially rolling back transactions. Each entry in the checkpoint log contains the following: (1) a list of shared objects whose read initiated the log entry, (2) a program location from where the transaction should proceed after a rollback to this checkpoint, and, (3) the current snapshot, called *continuation,* of the transaction's local data block; it is essentially a list of various undo-stack pointers.

2.2.2 Transaction Checkpointing

The default checkpoint – We associate a default checkpoint with every transaction in the system. Partial rollback to a default checkpoint equates to a transactional abort and full restart. Note that there is no cost associated with the default checkpoint creation.

What are the candidate checkpoints - In CCPR, we consider the first read operations on the shared objects as candidate operations for checkpoints. The reasoning behind this proposition is as follows: Each transaction in the system speculatively executes using a local data block. The actual shared objects are lazily updated during the transaction's commit operation. While a transaction is locally executing, some other transactions may commit, and hence, some or all of the shared objects that were read by this transaction may get updated. In such a case, this, not yet completed transaction, that had read the old values of the updated shared objects, becomes inconsistent, and needs to rollback to the first point, where the value of any of these shared objects were first read from the shared memory. Thus, the first read operations on the shared objects are candidate checkpoints in a transaction.

When to create a new checkpoint - Upon encountering a candidate checkpoint, a transaction needs to decide whether or not it actually needs to create a fresh checkpoint at the current operation. This decision is based upon the following factors:

- An executing transaction creates a new checkpoint at the read of a shared object only if the conflict probability value of the shared object is greater than some *threshold* value.
- Further, a transaction creates the new checkpoint only if the number of operations done by the transaction between this and the previous checkpoint is greater than some desired number of operations between two checkpoints.

When to cluster with an existing checkpoint - Once the above two factors are examined, the decision of whether to cluster with existing or to create a new checkpoint becomes clear.

How to create a new checkpoint - For all the local/shared objects in its local data block which have their dirty-bits set, the transaction pushes their current values in their respective undo-stacks and resets their dirty-bits. It then captures the current *continuation,* which in our framework is the current values of the various undo-stack pointers in the local data block, and, the transaction's program location. Subsequently, it creates an entry in the transaction's checkpoint log.

How to cluster with an existing checkpoint – Clustering a candidate checkpoint with a previous one just involves updating the victim shared object list of the previous checkpoint with the current shared object.

2.2.3 Conflict Probability

Initially when there are no transactions, the conflict probability of all the shared objects is 0. As multiple transactions read/write a shared object, they update its conflict probability value as follows.

Let us define the following:

k: total number of transactions accessing (reading and writing) the shared object.

n: total number of transactions that will update the shared object and thus will generate conflicts with other concurrently accessing transactions.

The conflict probability for the shared object is then set to ***n/k***, whenever ***k>1***.

2.2.4 Partial Rollback

Once a transaction is ready-to-commit, it needs to check whether or not it had read and used a consistent set of shared objects, which it does using the global version numbers as in the gv algorithm. Additionally it now needs to un-sync those shared variables in the shared object store, which it had read, and which are no longer consistent (if there are any of those). Finally, depending upon whether or not it was found safe to commit this transaction, a partial rollback operation is initiated. A partial rollback operation involves identifying the safest checkpoint to unroll the transaction to. The safest checkpoint is the earliest transaction program location, where the transaction read any of those shared objects which are out-of-sync now. The transaction progressively searches through the checkpoint log entries till it finds the first log entry pertaining to any of these out-of-sync shared objects, this entry is then considered as the safest checkpoint to unroll. Subsequently, the transaction applies the selected checkpoint's continuation and then proceeds from the rolled back transaction program location.

3. SIMULATION EXPERIMENTS

We developed a CCPR simulator to assess the overheads vs. conflict cost savings achievable through the algorithm. Two applications were studied in the experiments - Skip-Lists (SL), Red-Black-Trees (RBT). Insert and delete operations on these data structures were manually modeled as STM transactions, each transaction essentially being a series of read, write and other (e.g. comparison) operations on some shared/local variables. All experiments were done on an Intel dual-core machine. We varied the percentage of conflicting transactions by gradually increasing the number of transactions scheduled concurrently by the simulator

RBT – The application Red-Black Tree is a commonly used data structure. 500 random insert and delete RBT transactions were made to run on the simulator, and the total number of shared memory read operations performed for CCPR and for GV were counted. Figure 3 shows the savings in the conflict costs achievable through the CCPR algorithm. It compares the SMR_CCPR(i.e. total shared memory read operations in CCPR) with SMR_GV(i.e.. total shared memory read operations in GV). It is worth noting that as the percentage of conflicting transactions increases, the difference between SMR_CCPR and SMR_GV becomes more prominent. SMR_Defined is the actual number of shared memory read operations over all the transactions.

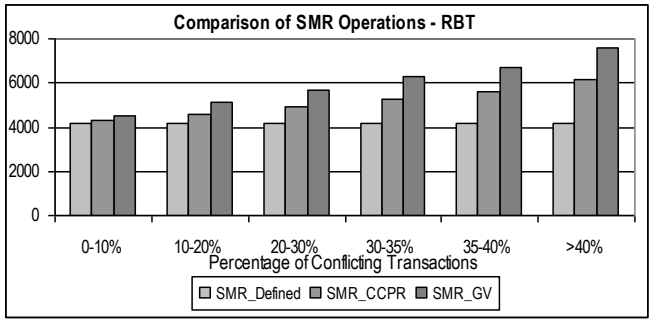

Figure 3. Comparison of shared memory read operations – RBT.

Figure 4 shows the effect of threshold probability on the percentage savings of the shared memory read operations. It is interesting to note that for this application high threshold probabilities of 0.6 and 1 were also good enough to considerably save on the conflict costs. Another thing worth noting is that, higher the threshold probability value, the lesser will be the overheads in our system, since most candidate checkpoints will be clustered with previous ones.

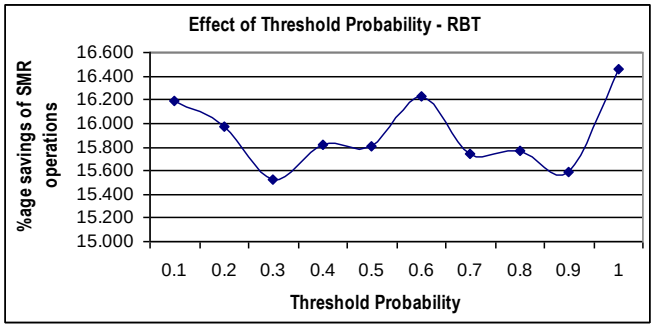

Figure 4. Effect of threshold probability on conflict cost savings - RBT.

Figure 5 and 6 characterize CCPR overheads in terms of checkpoints created by various transactions. It is interesting to note that as the percentage of conflicting transactions increase, on an average more number of checkpoints are created and when this percentage is less, transactions take lesser checkpoints, thus keeping the overheads under control.

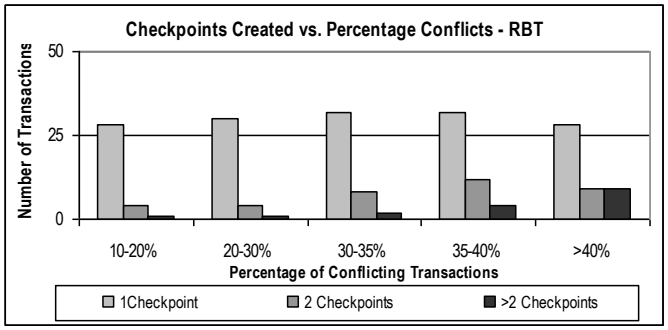

Figure 5. Effect of percentage of conflicting transactions on number of checkpoints taken – RBT.

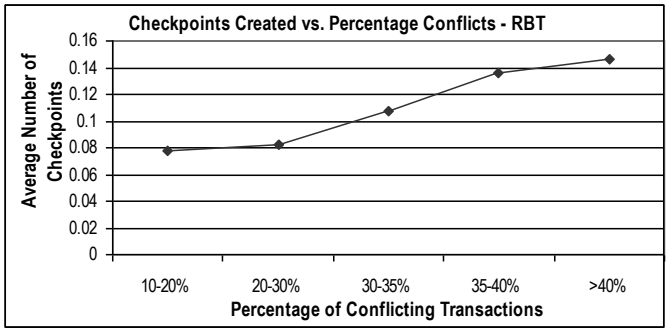

Figure 6. Effect of percentage of conflicting transactions on average number of checkpoints taken – RBT.

SL – The application Skip-Lists is another commonly used data structure. 500 random insert and delete SL transactions wete made to run on the simulator. Figures 7-10 present graphs similar in those of RBT.

It is worth noting that in this case a threshold probability of 0.4 was good enough.

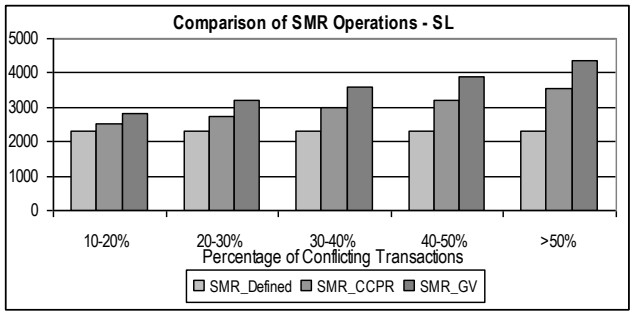

Figure 7. Comparison of shared memory read operations – SL.

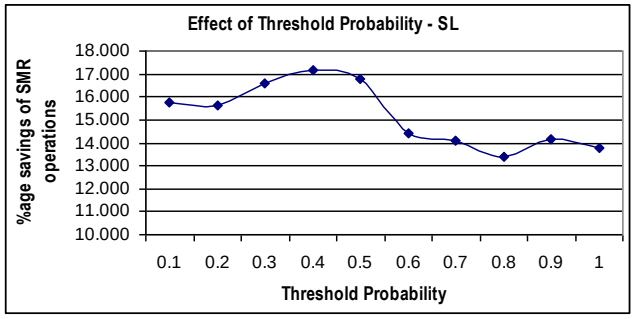

Figure 8. Effect of threshold probability on conflict cost savings - SL.

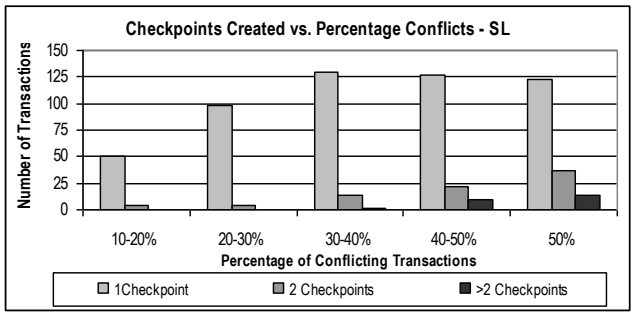

Figure 9. Effect of percentage of conflicting transactions on number of checkpoints taken – SL.

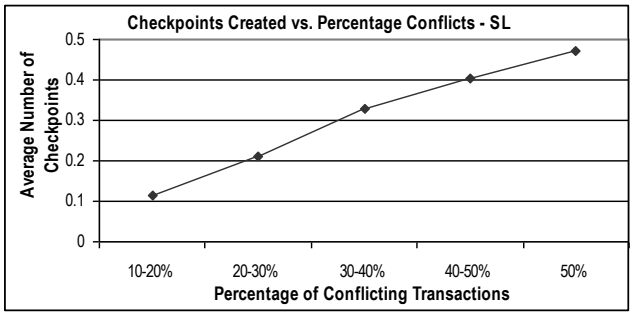

Figure 9. Effect of percentage of conflicting transactions on average number of checkpoints taken – SL.

4. RELATED WORK

Among existing non-blocking algorithms, TL2 [4] uses lazy versioning with commit-time locking. However, it is based on a global version-clock based validation technique, and does a lazy conflict detection followed by a full transaction abort if required. The CCPR algorithm in comparison proposes the use of continuous conflict detection with partial rollbacks if required.

Koskinen and Herlihy [5] first illustrated the use of checkpoints to do a partial rollback operation in boosted transactions [6]. Their work complements but does not completely replace conventional read/write synchronization and recovery. Our CCPR algorithm however, provides a full read/write synchronization and recovery technique based on automatic checkpointing, partial rollback and continuous conflict detection.

Waliullah and Stenstrom [7] suggested the use of checkpoints and partial rollbacks in the context of HTMs. Our proposal is with reference to STMs rather than HTMs and further the other main differences are: (1) The algorithm in [7] is demonstrated on a TCC framework which uses lazy conflict detection, as compared to the continuous conflict detection that CCPR uses, (2) In their algorithm, whenever a transaction commits, all addresses in its write set are compared with the read set of each of the ongoing transactions, and if a match is found, a conflict is generated. This method of conflict detection is very costly since each active transaction irrespective of whether or not conflicts, needs to be interrupted and checked during any other transaction's commit. CCPR does not have any such limitation. (3) Some of our suggestions to reduce CCPR overheads (e.g. moving from 1-CCPR to n-CCPR) can be applied in their framework to reduce their algorithm's overheads.

Other uses of the partial abort/rollback operations were mostly done [8, 9, 10] to provide support for open and closed nested transactions. However, these works differ from our work, since we use the concept of partial abort/rollback for undoing some operations within a transaction and hence do not require transactions to be nested to allow them to rollback partially.

Tabba et al. [11] propose a non-blocking, zero indirection transactional memory that can also use some HTM features for performance. They have a heavy usage of contention managers, are abort-centric and they show that they perform better for benchmarks with smaller number of conflicts. Unlike their

scheme, we have shown a good performance for cases with high conflicts.

Another line of work [12] in TMs is that of feedback directed dynamic selection of various implementations of atomic blocks. They are able to show that they reduce the wasted effort in aborted transactions by switching between optimistic and pessimistic concurrency control based on variables that can cause large number of conflicts. We have a probabilistic model that builds in these features in terms of determining the points to which a transaction should rollback and when should it rollback.

There has been good amount of work going on in building theoretical foundations for TMs as well. The works reported in [13], and [14] build formal models for STM and verification respectively. Recently, there have been a couple of studies on trade-offs involved in some important aspects of a TM. The work by Keidar and Perelman [15] studies the number of aborts that take place in a STM implementation and try to study the trade-off involved in reducing the number of aborts that are unnecessary. Another work by Attiya et al [16] studies the tradeoff in disjoint parallel access and indivisibility of read operations. These works shall be useful for future extensions and formal reasoning of CCPR.

5. CONCLUSIONS

We presented a novel partial rollback STM algorithm, CCPR, for intelligently checkpointing and partially rolling back transactions.

- Our simulation results establish that partially rolling back transactions is clearly desirable over full transactional aborts, especially when the percentage of conflicting transactions is high.
- Checkpointing a transaction saves a good amount of work that had to be done otherwise, in case of a conflict.
- Intelligent clustering of checkpoints helps reduce CCPR overheads and make it prone to cases when transactions are small or don't conflict much with each other.
- Further, it shows that CCPR can deliver a cost reduction of 16% to 18% in terms of reduction in the total number of shared memory read operations.

6. REFERENCES

[1] Nir Shavit and Dan Touitou. 1995. Software transactional memory. In PODC'95

[2] Larus, J.R. and Rajwar, R. 2006. Transactional Memory. Morgan and Claypool.

[3] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, Siddhartha Chatterjee. 2008. Software Transactional Memory: why is it only a research toy? DOI=http://www.cse.ust.hk/~charlesz/comp610/paper/p46-cascaval.pdf

[4] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. *DISC*, 06.

[5] Eric Koskinen and Maurice Herlihy. Checkpoints and continuations instead of nested transactions. In SPAA, 08

[6] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In PPoPP, 08

[7] M. M. Waliullah and P. Stenstrom. Intermediate checkpointing with conflicting access prediction in transactional memory systems. In IPDPS 08.

[8] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in logTM. SIGPLAN Not., 06.

[9] Yang Ni, Vijay S. Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *PPoPP*, 07.

[10] Tim Harris and Srdan Stipic. Abstract nested transactions. In TRANSACT, 07

[11] Tabba, Fuad and Moir, Mark and Goodman, James R. and Hay, Andrew W. and Wang, Cong. NZTM: nonblocking zero-indirection transactional memory. In SPAA' 09

[12] Sonmez, Nehir and Harris, Tim and Cristal, Adrian and Unsal, Osman S. and Valero, Mateo. Taking the heat off transactions: Dynamic selection of pessimistic concurrency control. In IPDPS '09

[13] Michael L. Scott. Sequential Specification of Transactional Memory Semantics. In *TRANSACT*, 06.

[14] Ariel Cohen, JohnW. O'Leary, Amir Pnueli, Mark R. Tuttle, and Lenore D. Zuck. Verifying Correctness of Transactional Memories. In *FMCAD*, 09

[15] Keidar, Idit and Perelman, Dmitri. On avoiding spare aborts in transactional memory. In SPAA 09

[16] Attiya, Hagit and Hillel, Eshcar and Milani, Alessia. Inherent limitations on disjoint-access parallel implementations of transactional memory. In SPAA 09.