Are Database-style Transactions Right for Modern Parallel Programs ?

Jaswanth Sreeram
College of Computing
Georgia Institute of Technology
jaswanth@gatech.edu

Santosh Pande
College of Computing
Georgia Institute of Technology
santosh@cc.gatech.edu

Abstract

The early transactional memory (TM) programming model and semantics were inspired by ideas and semantics from database transactions and today's state-of-the-art TM systems are not too removed from them. But today's TM systems are being used in modern and emerging parallel applications which are very different from the database programs that those transaction semantics and programming model were designed for. The differences include the nature of the synchronization itself, the amount of flexibility demanded from the programming model and the amount of programmer control desired in the programming process. While todays TM systems offer an elegant and concise conceptual interface they typically offer a rigid set of semantics to the programmer and are intended to be used only as black-box components while writing parallel programs. They provide guarantees of properties such as strict atomicity and isolation irrespective of whether a program's semantics require them and usually at a significant performance cost. In this paper we argue that this rigid view of memory transactions limits the range of expressibility necessary for supporting some important parallel programming idioms for some soft-computing programs. Using three specific behaviors as examples we also argue that relaxing some of these guarantees and maintaining the flexibility to support the specific style of synchronization a program needs can substantially improve expressibility and parallel performance especially in programs with long-running transactions which discard significant amounts of work when they abort.

1 Introduction

The widespread popularity of multi-core processors has made it necessary to provide programmers with programming models that enable them to develop parallel programs that are both correct, efficient and scalable. The Transactional Memory (TM) model [10] has been widely studied and is touted as an elegant abstraction to express data synchronization. Such synchronization is expressed via specifying *atomic* blocks of code which are guaranteed to execute atomically - each atomic block of code appears to execute at once in during some indivisible instant of time. Therefore in contrast with fine-grained locks programmers using memory transactions can simply specify where atomicity is needed instead of also having to specify how to achieve it. This programmability advantage is the primary appeal of TM language extensions and systems.

Memory transactions were conceptualized from database transactions and they retain many of the traits of their database counterparts - guaranteeing ACID: strong atomicity, consistency, isolation and durability, separating atomicity from the method for achieving it and so on. However in our opinion database transactions capture very different computation than modern real-world parallel programs. DB transactions typically capture the business logic of commercial or enterprise workloads where the ACID properties above are desirable. Contrast this with a modern real world parallel program such as a state-of-the art parallel game engine. It is not clear that simply using analogues of database transactions to manage data synchronization in such an application is a prudent idea. Real DB transactions are oriented around inserting, querying, deleting records and performing some relatively simple operations on the returned data. Moreover the data schema manipulated by the user is relatively simple - tables, rows and columns. Many critical regions in modern parallel programs however implement much more complex functionality such as constraint or equation solvers, physical simulation or some non-trivial algorithm. And these critical regions often turn out to have a significant influence on overall parallel performance. Furthermore in contrast with database transactions many of these critical regions have the programmer interact with complex data structures - e.g., a scene graph or voxel-octrees in graphics and interactive simulations. The oft-used standard database example of concurrent deposits and withdrawals from a bank account may be a good simple representative case for thinking about database transactions and their properties but it does not capture the complexity and diversity of behavior in modern general purpose parallel programs.

A simple conceptual and programmatic interface for specifying atomic sections in parallel programs is certainly useful and memory transactions fit this role. However using the database notions of atomicity, consistency and isolation as the sole basis for the transactional programming model limits the diversity of synchronization idioms that can be expressed using this interface. Consider the following three properties provided by a TM system:

1. **Atomicity**: All TM systems provide the *atomicity* guarantee. In many TM systems when a transaction reads state that has been overwritten by another concurrent transac-

tion that committed, the reader transaction is aborted and restarted. This is automatically guaranteed without regard to whether it is desirable in the context of the program's semantics. Of course, this guarantee is important for the correctness of many programs (indeed this property is extremely well aligned with the semantics desirable in common database applications) but for others it may be unnecessary and even undesirable.

2. **Isolation**: A transaction does not have any knowledge of other concurrent transactions. In combination with the *atomicity* property above, this means that the TM model dictates that the reader transaction should abort regardless of which specific writer transaction performed the update, even if such behavior is not required by the programs semantics.

3. **No user involvement**: Some TM systems allow the user or the programmer to provide annotations or hints to turn on or off specific behaviors or algorithms in the TM runtime such as log compaction, eager or lazy conflict detection, commit-time or encounter-time locking etc. The expectation is that the programmer has the best knowledge of which of these options is most suitable for his program and will supply the annotations appropriately. However most TM systems do not allow the programmer to specify behavior such as specifying meaningful actions on important events like aborts and commits or see the transaction's state. As far as the program is concerned the state maintained by the transaction itself is off-limits. There are good reasons for this limitations, two of them being preserving programmability and preserving portability between different TM systems. So while this limitation makes it easier for novice programmers to reason about synchronization, it also severely limits what kinds of semantics other programmers can express in their transactions.

In this paper we argue that relaxing these properties is meaningful in some programs and that the apparent simplicity of using database style transactions does not necessarily make expressing complex semantics in some modern parallel programs easier. In the following sections we describe three behaviors that commonly occur in parallel programs that are not easily captured by the traditional notions of memory transactions in that they require some violation of the strict notions of atomicity, consistency, isolation or require user involvement.

2 Relaxed Sharing

There is a large class of real-world programs called *soft computing applications* which are characterized by several unique properties [4].

- **Approximate nature of results.** These applications all produce an approximation of the actual results rather than their actual values. This may be because of several reasons. One common reason is that the physical or mathematical model expressed in the program requires some approximation to be computable in a reasonable amount of time. Other programs such as simulation applications mimic continuous processes but in a discrete-time fashion and this introduces some error in the result.

- **User-defined correctness.** In some cases, the application programmer can choose to consciously sacrifice accuracy of the results in order for the program to meet some execution characteristics such as soft real-time deadlines. He or she may be able to control parameters that directly determine the amount of error in the results produced. Examples of such parameters include thresholds in approximations, the granularity of ticks in time-stepped simulations, cutoff distances and radii in physical simulations etc.

- **Tolerance for Imprecision and Uncertainty.** Soft computing applications to some extent are tolerant of imprecision in inputs and some program values. Many such applications are designed to work with input streams and program values which are inherently noisy, imprecise or unreliable. Examples of such programs include pattern recognition systems, object-tracking systems and other machine learning applications.

In many such soft computing applications [12, 9, 18, 11], the values produced into shared variables in critical sections, undergo transformations that change them very little in relative terms [8, 5, 6, 2]. Relatedly they also exhibit the *Approximate Store Value Locality* phenomenon [4] - a significant portion of writes to shared variables in critical sections end up writing a value that is very close to the value already present at that memory address. The *closeness* of the values before and after the write is of course

Some reasons for this behavior in these programs are:

- **Similarity in input data**: Many real-world input data sets contain a substantial number of input values that are similar.

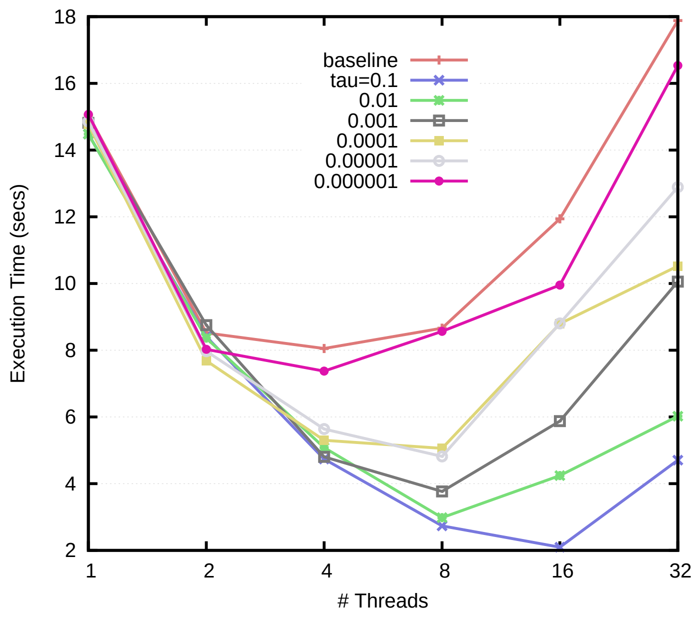
- **Iterative refinement**: Many critical sections occur inside loops where the results computed in the loop body are synchronized with the global state at the end of each iteration. If the results computed are similar or approximately similar for two consecutive iterations (i.e., each thread, modifies global state by a relatively small magnitude), then the store in the critical section that updates global state will often exhibit the above store locality property.

- **Finite precision in hardware**

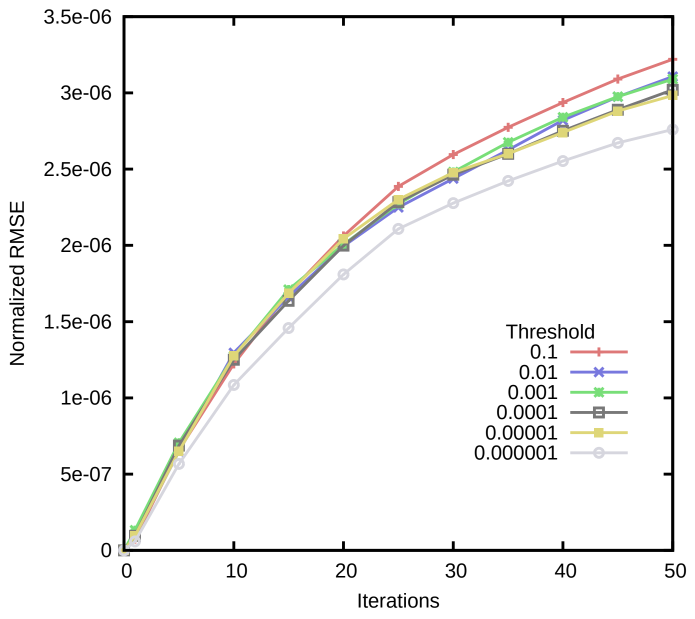
2.1 Exploiting Approximate Value Locality

If consecutive updates made to the shared state by a writer transaction are relatively small, then the reader may be able to proceed with the older state without waiting for the newest value, as happens in normal (precise) synchronization in TMs. This behavior usually constitutes a violation of the atomicity property (it may not violate atomicity if the reader does not make any subsequent modifications to shared state based on the result of this read). Whether this violation is safe or desirable is dictated by the program semantics for example, the "approximate results" properties of soft computing applications above may be enable them to tolerate this loss of atomicity.

Consider a transactional implementation of the `kmeans` algorithm [3]. This algorithm takes a set of objects and attempts to cluster them based on a distance metric. The program starts by assigning objects randomly to a set of initial clusters. Then



(a) Execution time using Relaxed Sharing. τ refers to the relative threshold



(b) Growth of error in results when using Relaxed Sharing. τ refers to the relative threshold

Figure 1: kmeans

Algorithm 1 `Kmeans`

```
while delta > 0 do
  delta = 0
  for all Object ``i'' do
    atomic {
    //Reads c→center
    //(Reader transaction)
    cc = findNearestClusterCenter(i)
    }
    if membership[i] ≠ cc then
      membership[i] = cc
      delta += 1
    end if
  end for
  for all Cluster ``c'' do
    atomic {
    //Writes c→center
    //(Writer transaction)
    c→center = computeNewCenter(c)
    }
  end for
end while
```

the new cluster centers are computed by summing the objects within each cluster. These centers are computed and stored atomically in a transaction as shown in Algorithm 1. In the next iteration the distance of each point in a partition from all the cluster centers is computed. For a random distribution of initial cluster centers and objects, the relative amount of change in the position of the cluster centers is quite small over successive iterations. Therefore, we can exploit the approximate store locality in the cluster centers and ignore updates to the cluster center that are small. This can be implemented by applying an approximate locality threshold τ to the shared variables holding the positions of the cluster centers. Consider a thread A that has read the position of a particular cluster center in order to compute its distance from objects in A's partition. Now the thread B that owns this cluster center computes a new cluster center which may be less than τ away from the current cluster center that A has read. Therefore the store executed by B is an *approximately local* store and would be marked as such. When thread A finishes computing the distances of each of its objects from the old cluster center these distances may be inconsistent. However if the relative magnitude of this inconsistency is small A can go ahead with the next step of reassigning objects instead of aborting and restarting. The parallel performance benefits of applying such a relaxed sharing technique for kmeans using the TL2 [7] STM are shown in Figure 1 (a). The baseline refers to an unmodified version of the program and the τ refers to the relative threshold at which two values are considered equal. That is, if a transactional store in a writer transaction writes the value v_1 to a shared memory location presently containing the value v_0 then the new value v_1 is ignored by the reader transaction if $|v_0 - v_1|/v_0 < \tau$. Thus a higher τ represents a higher degree of relaxation in the sharing and vice versa. The plot in 1 (b) shows the growth of the normalized root mean square error (denoted by RMSE on the Y-axis) in the cluster centers computed by the threads over the program's execution (denoted by the iteration number in the X-axis). This shows that while relaxed sharing of values introduces an error into the computation and this error accumulates, this *error still grows relatively slowly and smoothly over the execution of this program.*

3 Fine Grained Consistency Rules

The consistency property in database and memory transactions guarantees that all the shared variables read in a transaction are consistent as according to some serializable schedule of all the transactions in the system. However in some programs such consistency may be required only on a specific set of variables. That is, some sets of variables are required to be consistent and the others variables accessed in the transaction are not. Consider the example of a game engine that models a set of movable objects (players, weapons, vehicles, projectiles, particles, arbitrary objects etc). Each of these game objects is represented by a program object that has among others, three mutable fields representing x,y,z positions of the object at an instant. The game object can be subject to many factors that change its position - game play factors like user input, movement due to being attached to other bodies in a joint, physical forces like collision with another body and so on. The program object representing this game object is shared among all the modules implementing those forces. This program object is (or atleast the fields in that object are) thus potentially touched by a very large number of writers. It is also accessed by a large number of readers. For example, the rendering engine reads the position fields in order to perform the visibility test and to draw the object into the graphics frame-buffer. Other readers of these fields could include physics modules that perform collision detection, and scripts that trigger events based on the players proximity. However the position fields need not be accurate on every frame and all the readers do not need the most up-to-date values to execute correctly. For example, reading accurate position values in collision detection may be more important than in triggering events like special effects. Additionally, the modifications made by all writers are not equally important and some modifications can be safely ignored. For example, minor modifications to a moving particle's position due to wind or gravity can be safely ignored from frame to frame. Such semantics are at best clumsy or at worst not possible to express with current TM programming models. Below we outline one sample mechanism that can extend the TM programming model to allow a programmer to specify such semantics.

3.1 Variable Groups

The shared variables that can be accessed transactionally are divided into abstract *consistency groups*. The assignment of a particular variable to a group can be dynamic (during the lifetime of a transaction) and the same variable can be assigned to multiple groups at once. A consistency group can subscribe to a set of *Consistency Rules* that describe the valid operations that can be performed variables in the group without violating *abstract consistency* or the consistency required by the semantics of the program. For example, the x,y,z position fields inside a reader transaction implementing visualization in the above example may be placed in a consistency group that has consistency rules that permit a particular writer transaction implementing wind or gravitational forces to modify the fields without forcing an abort on the reader transaction. The same consistency group can also subscribe to a separate consistency rule that ignores writes from a transaction implementing collision detection of this body with inactive scenery or with objects that do not transfer momentum. Our preliminary experiments with such fine-grained consistency control in a interactive particle simulation engine showed that with these enhancements, transactions in this program can achieve a peak throughput (rate of transaction commit) of 96% of the theoretical maximum peak throughput (transactions executing without any concurrency control).

Several of these optimization "tricks" are prevalent on modern game engines and it is desirable that the choice of the synchronization and concurrency control mechanism does not prevent the ability of programmer to use them or make such semantics prohibitively cumbersome to specify.

4 User specified conflict handling and recovery

A (reader) transaction that reads some value is said to experience a conflict on that value when a concurrent writer transaction updated that value and then committed. The reader transaction can detect such a conflict during the read itself, at some point later in its execution or when it attempts to commit. Irrespective or when this conflict is detected, in most current TM programming models the reader transaction has no choice but to abort (see [20], [24], [22] and [21] for proposals that suggest alternate options). This means that all the work performed so far in the transaction is discarded and the transaction starts again. For some transactions this can be avoided if the results and the state of the aborting transaction can be `repaired` in which case this transaction can instead "roll-forward" and proceed with its execution (or re-attempt to commit if the conflict was detected when it attempted to commit).

A simple example of a transaction that performs a key lookup on a list is shown in Figure 2. The `tm_read` and `tm_write` calls signify transactional reads and writes of the variable specified.

Let transaction T1 be executing the code in Figure 2(a) while a concurrent transaction T2 is modifying the list for example by inserting new elements into it. During validation T1 will detect a conflict if it has read `p→next` for some node p and T2 has modified `p→next` to point to a newly inserted node q (and committed). Instead of aborting at that point T1 can use the new value of `p→next` and attempt to continue the search from q. This is specified in the recovery action specified by the user with the `OnConflict` primitive. This primitive operates as follows. When a conflict is detected the TM runtime returns the new updated value to the application program in the container attached to the `OnConflict` primitive - in this case `n`. Then the transaction jumps to the statement immediately after the declaration of the `OnConflict` primitive and starts executing with this new value of `n`. At a high-level this recovery action says that when a new node is inserted after the node pointed to by `n` resulting in a conflict on `n→next`, the transaction can recover by resuming the lookup at the new node now pointed to by `n→next`. In this way T1 can re-use the intermediate result of the computation for finding the key in the list upto node p. This recovery action also enables the transaction to recover from node deletions. Additionally, multiple conflicts

```
atomic {
 node_t *n = list->head;
 for(;n;) {
  if(n->key == key)
   break;
  n = tm_read(n->next);
 }
}
```

(a) Original

```
atomic {
 node_t *n = list->head;
 OnConflict(n) {
  for(;n;) {
   if(n->key == key)
    break;
   n = tm_read(n->next);
  }
 }
}
```

(b) With a user specified recovery action

Figure 2: List search

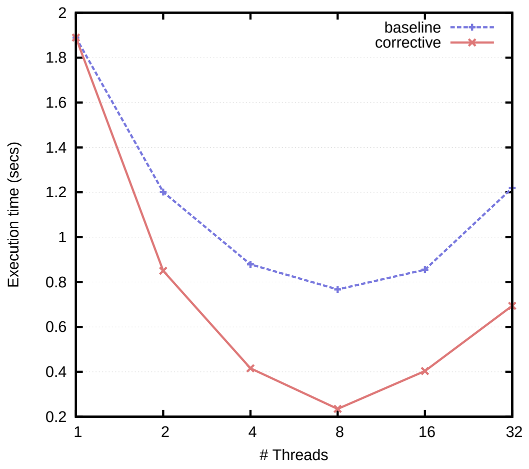


Figure 3: Execution time of the list search in Figure 2 with and without the corrective recovery action.

can also be handled seamlessly - the runtime detects one conflict at a time and each of these conflicts can trigger the recovery action in turn. Note that the transaction did not have to be modified substantially to accommodate this recovery action. Yet this action transforms the simple lookup transaction into a transaction implementing a non-trivial *incremental lookup algorithm*. We implemented a simple synthetic `list` benchmark that performs transactional lookups, inserts and deletes. The baseline version of this program uses the TL2 STM system. We then implemented the recovery action described above in the lookup algorithm and compared its parallel performance to the baseline version. This plot is shown in Figure 3.

While this example consisted of small transactions, for large long-running transactions, the performance advantages due to such recovery actions could improve overall performance even more significantly. The usage the recovery action above required that the transaction reveal some of its internal state to the application - specifically the node at which the conflict occurred and the new value written by the other transaction. This is not permitted by current TM programming models as a matter of both programmability as well as portability. In some cases a programmer familiar with the algorithm may be able to specify such recovery actions relatively easily but for complex transactions this might be more challenging. But for simple enough transactions that experience high-contention, the loss in programmability is well worth the gain in parallel performance. As for portability of recovery actions, a standardized interface across TM models can alleviate this problem (the portability issue is akin to the similar issue for implementing open-nesting with clear consistent semantics across various TM platforms).

5 Related Work

Techniques such as open nesting [15, 16] and the Galois system [17] have been studied extensively. At a high level, the goal of both these models is to separate physical conflicts from semantic conflicts since usually only the latter is required for correctness. Therefore strict physical serializability is traded for *abstract serializability*. Abstract Nested Transactions [24] allow a programmer to specify operations that are likely to be involved in benign conflicts and which can be re-executed. The notions of Load Value Locality [2, 14] and Frequent Value Locality [13] have been studied extensively. Load value locality refers to the observation that values accessed by loads and stores have a repetitive nature to them and techniques such as *value prediction* exploit this locality to apply optimizations such as cache prefetching. Frequent value locality analysis on the other hand describes the values which collectively form the majority of values in memory at an instant in program execution. Previous works such as [1] have investigated the problem of multiple consistency and isolation levels in database transactions. In [19, 23] the authors explore the performance vs accuracy trade-off by controlling the floating point precision in a real-time physics engine. These techniques are closely related to the relaxed sharing and fine-grained consistency rules described above. In [21] the authors describe a distributed database transaction model in which conflicting database transactions can be reconciled and allowed to commit in a specific order. Much like the user-defined recovery actions described in Section 4, this technique also relies on specialized reconciliation actions to repair the state of the distributed database.

6 Conclusions

In this paper we ask the question whether the database-style TM semantics and programming models being studied today are appropriate for modern, emerging parallel applications. Specifically we discussed three phenomena ”Approximate Sharing”, ”Fine-Grained Consistency” and ”User defined recovery” that we feel are important to such programs and we discuss the difficulty of using TM programming models to express these idioms. Programmability, while desirable, should not necessarily result in restricting the kinds of semantics that can be expressed in parallel programs. Several research issues remain to be explored in this direction in order to answer the question posed but we hope that this work will serve to provoke further investigation.

References

[1] Gray, J. N., Lorie, R. A., Putzolu, G. R., and Traiger, I. L. 1988. Granularity of locks and degrees of consistency in a shared data base. In Readings in Database Systems Morgan Kaufmann Publishers, San Francisco, CA, 94-121.

[2] Lipasti, M. H., Wilkerson, C. B., Shen, J. P. Value locality and load value prediction. SIGOPS Oper. Syst. Rev. 30, 5 (Dec. 1996), 138-147

[3] Minh C. C., Chung J., Kozyrakis C., Olukotun K., STAMP: Stanford Transactional Applications for Multi-Processing. IISWC 2008: 35-46

[4] Sreeram J., Pande S., Exploiting Approximate Value Locality for Data Synchronization on Multi-core processors. IISWC 2010

[5] Richardson S.E., Exploiting Trivial and Redundant Computation, Sun Microsystems Technical Report 1993.

[6] Citron D. and Feitelson D. G.,"Look It Up" or "Do the Math": An Energy, Area, and Timing Analysis of Instruction Reuse and Memoization, in Workshop on Power Aware Computing Systems 2004.

[7] Dice D., Shalev O., Shavit N., Transactional Locking II. Proceedings of the 20th International Symposium on Distributed Computing (DISC), 2006.

[8] Richardson S. E., Caching Function Results: Faster Arithmetic by Avoiding Unnecessary Computation Sun Microsystems Technical Report TR-92-1 1992.

[9] Zadeh L.A., Fuzzy logic, neural networks, and soft computing. Communications of the ACM, 37(3):7784, 1994.

[10] Shavit, N., Touitou, D. Software transactional memory. PODC 95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing (New York, NY, USA, 1995), ACM Press, pp. 204213.

[11] De Gelas, J. The quest for more processing power: Multi-core and multi-threaded gaming. http://www.anandtech.com/cpuchipsets/showdoc.aspx, March 2005.

[12] Baek W., Chung J., Minh C. C., Kozyrakis C., and Olukotun K., Towards soft optimization techniques for parallel cognitive applications. In Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures (San Diego, California, USA, June 09 - 11, 2007). SPAA '07. ACM, New York, NY, 59-60.

[13] Yang, J. and Gupta, R. 2002. Frequent value locality and its applications. Trans. on Embedded Computing Sys. 1, 1 (Nov. 2002)

[14] Lepak, K. M., Exploring, Defining, and Exploiting Recent Store Value Locality. Ph.D. thesis. The University of Wisconsin-Madison, Department of Electrical and Computer Engineering. Dec. 2003.

[15] Ni Y., Menon V., Adl-Tabatabai A. R, Hosking A.L, Hudson R.L., Moss J.E.B., Saha B., Shpeisman T., Open nesting in software transactional memory. Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP):68-78 March 2007.

[16] Eliot, J., Moss, B., Open nested transactions: Semantics and support. In Workshop on Memory Performance Issues 2005.

[17] Kulkarni M., Pingali K., Walter B., Ramanarayanan G., Bala K. and Chew L. P. 2007. Optimistic parallelism requires abstractions. In Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (San Diego, California, USA, June 10 - 13, 2007)

[18] Sims K., Particle Animation and Rendering Using Data Parallel Computation. In Proceedings of SIGGRAPH 1990

[19] Yeh T. Y., Reinman G., Patel S. J., and Faloutsos P. 2009. Fool me twice: Exploring and exploiting error tolerance in physics-based animation. ACM Trans. Graph. 29, Dec. 2009

[20] M. Herlihy and E. Koskinen "Checkpoints and continuations instead of nested transactions", in the 3rd Workshop on Transactional Computing

[21] Phatak, S.H. and Badrinath, B.R. "Multiversion Reconciliation for Mobile Databases" in Proceedings of the 15th international Conference on Data Engineering 1999.

[22] Ramadan, H. E., Roy, I., Herlihy, M., Witchel, E, "Committing Conflicting Transactions in an STM", in Proc. of the International Symposium on the Principles and Practice of Parallel Programming (PPOPP) 2009.

[23] Yeh T., Faloutsos P., Ercegovac M., Patel S. and Reinman G. 2007. The Art of Deception: Adaptive Precision Reduction for Area Efficient Physics Acceleration. In Proceedings of the 40th Annual IEEE/ACM international Symposium on Microarchitecture (December 01 - 05, 2007).

[24] Harris T. and Stipic S., "Abstract Nested Transactions", in 2nd Workshop on Transactional Computing (TRANSACT 07)